Considerations underlying the design of the central processing unit are discussed.

Particular emphasis is placed on addressing, sequencing, and monitor control functions as well as provisions for arithmetic and logical operations.

The structure of SYSTEM/360

Part III – Processing unit design considerations

by G. M. Amdahl

SYSTEM/360 is a general purpose system for commercial, scientific, communication, and control applications. It is designed for implementation over a wide cost and performance range. In view of design objectives, a number of the system and processing requirements demanded special attention in respect to cost/performance and ease of programming. One set of hardware provisions often had to serve several needs. Seldom was a design solution adopted purely because of hardware economy; in general, attainment of complete and unified structure was the more important consideration.

The system and processing requirements range from the addressing, conditional branching, indexing, interruption, storage protection, and input/output (I/O) control needs to the less prominent but very vital needs for storage allocation, subroutine linkage, indirect execution, monitor control, bit manipulation, and code translation and recognition. Many of the provisions for these tasks, as well as some of the considerations behind them, are discussed.

The material is neither complete in terms of the special functions provided nor exhaustive in the supporting considerations which affected the solutions presented. However, it is indicative of the approach employed in defining and categorizing the special requirements, as well as in selecting and modifying the design solutions to match these needs.

Addressing

In planning the addressing structure of SYSTEM/360, many desirable characteristics were considered. The principal ones were:

- Binary addressing.
- Addressing to the byte.
- Ability to address a very large storage.
- Bit efficiency of programs.
- Efficient relocating loader.
- Use of shared programs.
- Feasibility of programming conventions for dynamic relocation.

binary addressing

For lookup purposes, binary addressing permits the use of arbitrary bit patterns as increments to table bases. Although lookup can be accommodated by addressing structures that utilize other radices, more manipulation is required in these cases. Other advantages of binary addressing accrue from the greater compression of address information and the greater resolution of truncated addresses (bytes, halfwords, words, double words, etc. can be readily specified).

byte addressing

The principal motivation for byte addressing appears in commercial application areas. In this use, records consist of many fields of widely differing lengths, where each length is normally defined in terms of the required information content. It is desirable that information be packed, not only to permit storage efficiency, but also for efficiency of transmission between the peripheral storage device and the computer. It is also preferable that the information appear in an order meaningful to humans who must communicate with a file.

It seemed to be desirable to enrich the character set available to the computer user and, at the same time, to compress information. At first thought, the two factors appeared incompatible. However, studies of commercial file statistics indicated that more than 60 percent of the characters in the average record were decimal numerics. Both purposes could then be served by selection of an 8-bit character, representing either a single non-numeric character or a pair of decimal digits.

A similar argument applies to the task of communication between the human and the machine. The least restrictive mode of communication from the human standpoint is the ability to provide and receive strings of symbols with starting and ending points that are arbitrary in respect to machine word boundaries. Communication with humans must certainly be expected to take on increasing importance. This strongly recommends that a computer be able to consider a string of symbols as an entity and to deal with the string wherever it resides.

Addressing to the bit (as in the IBM 7030) was considered because it permits use of direct techniques for manipulating arbitrary bit groups. The low frequency of these applications, compared to byte handling, as well as the requirements for three additional address bits and more hardware controls, made this

type of addressing unattractive. Efficient alternatives, described later in this paper, are provided in SYSTEM/360.

large capacity storage

The ability to address a large storage is becoming a more important characteristic of machines as the technology of storage devices advances. The use of large-capacity storage reduces the number of references to external storage devices, in part because much larger records can be accommodated and, in even greater part, because much larger problems can be contained throughout their execution. In file processing applications, several runs may be combined into one because the storage capacity permits retention of all programs for the many category, status, and action determinations involved. Even where access to external storage devices is required, the larger assignable buffer areas permit much more overlapping of access and processing.

Use of computers for new and more complex tasks becomes easier as storage capacity increases. Various important techniques, such as list processing, will only thrive in an environment of large immediate-access storage. Many applications can be more efficiently attacked by the utilization of large tables. Such activities occur in language translation and probably will be significant in speech- and pattern-recognition areas as well. In scientific calculations, large tabular functions also appear promising for more efficient equation evaluation.

With the advent of larger storage capacities, then, a substantial increase in processing is expected from the use of larger data arrays. Yet the number of individually named arrays and quantities is not expected to increase nearly as much. Adoption of an addressing technique more closely related to symbolic naming may be expected to serve both of these growth characteristics.

The desire for bit efficiency in programs persists despite the advent of large capacity storage. Exploitation of a structure containing such storage requires that programs and principal data areas reside in main storage and that supporting programs, data, and tables remain in the more-capacious but lower-speed storage, at least during periods of latency. Consequently, information compression should have beneficial effects on storage-to-storage transmission times as well as on storage utilization.

relocating loader

Loading of programs into computers is becoming an increasingly complex task. The programs loaded today are collections of programs, even though each program at its time of writing is viewed as independent of others with which it may ultimately combine. Neither can a program be written to occupy particular positions within storage, nor can the data addressed by the program be assumed to occupy relatively invariant positions. In most current machine structures, the relocation or assembly process involves inspection of all address quantities residing within instructions, branching tables, calling sequences, etc. The efficiency of this relocation procedure would be improved if addressing information were concentrated in two ways: (1) in

space, so that large spans of code need not be inspected instruction by instruction, and (2) in quantity, by avoiding much of the replication involved in storing a complete address for each separate reference to a region.

shared programs

Sharing a common copy of a program between several concurrent users is valuable in a number of application areas to which computers recently have been applied. These applications are primarily characterized by the use of a central computer to serve a number of on-line consoles, at each of which an operator may be handling a transaction in one of many predefined ways. Examples are airline reservations, credit investigations, and bank teller functions.

In these applications, the program segment traffic generally exceeds the data record traffic, although the space requirements for the program segments are usually less than for data residence. With the advent of larger storage devices, it appears feasible to retain complete copies of at least the heavily used service routines within core storage, provided that these copies need not be duplicated. Such a procedure would reduce system delays due to continued reference to peripheral storage devices of relatively long access time. Also, traffic for the programs, the most often sought information in the system, would be eliminated.

Such a technique can be effective only if the addressing structure of the computer can accommodate a programming convention that permits the code to remain unchanged throughout its execution. This requires that the addresses for all data and working areas currently allocated to a given user be available from the contents of the registers of the central processing unit (CPU).

dynamic relocation

Dynamic relocation of programs and/or data areas is a problem occurring in time-shared multiprogramming applications. In such an application, a given latent program may be displaced from storage either by the requirements of an active program or a program of higher priority. At a later time, it is desired to bring this program back into storage and to resume its execution. At this point, two alternatives exist: delay the return of the displaced program until its previous residence areas become available, or relocate this program to regions of storage currently available. The former alternative involves further delay for the displaced program; the latter alternative requires that all address-like information be identified and appropriately modified. It is easy enough to define an automatic dynamic relocation addressing technique which entails considerable penalty in either hardware or time, or in both. The challenge is to provide an addressing technique whose inherent characteristics permit the adoption of programming conventions that are capable of programmed relocation with reasonable efficiency.

base addressing

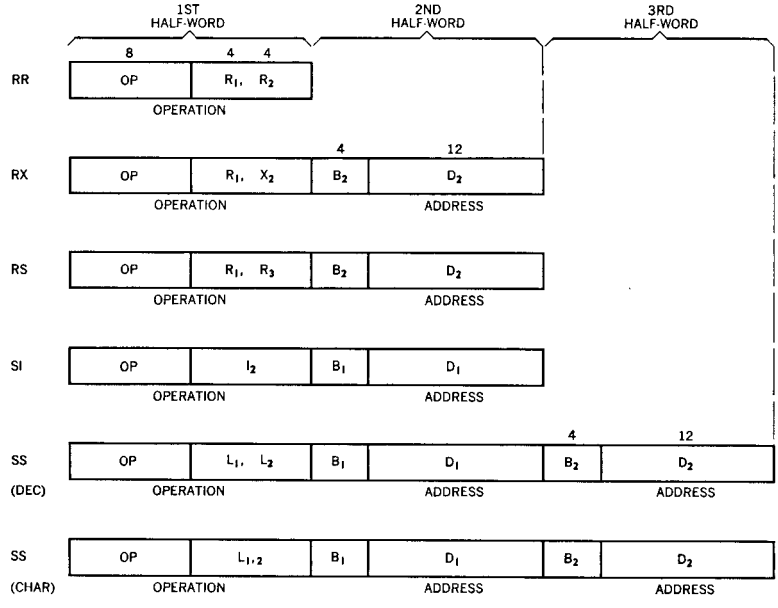
After much consideration of the foregoing desired characteristics in the computer's addressing structure, a modified form of base addressing was employed. The base addresses for address

computations are supplied from registers. Two additional properties serve the structure for greater efficiency and ease: (1) a displacement quantity in the instruction and (2) an additional register reference for greater generality in address generation. As indicated in Figure 1, the format chosen employs the address syllable. This format is adhered to in all CPU instructions. The I/O commands, however, contain absolute addresses because, by virtue of their asynchronous execution, they can have no access to relevant CPU register contents.

The displacement field in the instruction must be adequate for the vast majority of relative addresses. On the other hand, undue length would encourage use of the displacement for absolute addresses. It would also encourage the consolidation, before loops are entered, of outer index values with the displacement. Such practices would thwart the end objectives of the addressing philosophy. A displacement field of twelve bits was chosen as an effective compromise.

A second register address appears in the formats of those instructions that are normally employed in processes involving regular progression through arrays of data. This provision of a second register address is called *double indexing*. Double indexing makes it possible to reduce register traffic in two ways. First, if bases are separated from indices, a load instruction suffices to replace an unmodified base quantity by another base; were the two combined, a store would be needed to preserve the previous value. Secondly, the number of values to be updated in a loop iteration is limited by the given number of increment sizes, rather than by the sum over arrays of increment sizes for each array involved. Double indexing is provided in storage-to-register arith-

Figure 1 Instruction formats



metic and logical operations and in branches, excepting the indexing branches and the multiple-register loads and stores. Multiple-register loads and stores are seldom invoked in array processing.

To further improve the indexing and addressing capabilities of SYSTEM/360, the set of fixed-point arithmetic accumulators was chosen to serve jointly for index registers and base registers. The full power of the fixed-point arithmetic functions was thus made immediately available for address and index computations. Moreover, the results of logical and fixed-point arithmetic operations were made directly accessible to the addressing mechanism for use in data lookup or sequencing functions.

Sequencing

The sequential control of a computing system should provide for:

- Conditional branching on the outcome of logical or arithmetic tests.
- Return address and/or calling sequence information (subroutine linkage).
- Abrupt change in sequence control to special routines in the event of unusual system conditions (interruption).
- Indirect execution of instructions.

conditional branching

Excepting the index branches, the SYSTEM/360 branch instructions appear in both the one-syllable and two-syllable formats, the two-syllable format permitting double indexing of the branch address.

In SYSTEM/360, the BRANCH ON CONDITION instruction operates in conjunction with a condition register (CR). The CR is variously set to one of its four states in the course of arithmetic or logical operations. For *and*, *or*, or *exclusive or*, conditions are set to indicate either a complete zero result or a non-zero result. In the case of compares, one of three conditions is set to indicate low, equal, or high. For tests of register contents, again one of three conditions is indicated: negative, zero, or positive. Finally, for add and subtract, one of four conditions is set: negative, zero, positive, or overflow. No conditions are set for multiply, divide, load, or store; it is desired that these operations be performed without disturbing the current value of the CR and, moreover, result conditions can be predicted from initial operand tests. The BRANCH ON CONDITION format contains a 4-bit tag, one bit position for each CR state. The CR states effecting the branch are specified by 1's in the bit positions corresponding to selected states. If the CR is in a state for which there is a corresponding 1 (0) in the condition tag, the branch occurs (does not occur). Thus, all combinations of states may be symmetrically defined as branch or no-branch situations. Note that 1111 in the condition tag corresponds to an unconditional branch, for a branch occurs on any CR state. On the other hand, with 0000 in the condi-

tion tag, no branch occurs regardless of the CR state, thus effectively providing a "no operation" instruction. BRANCH ON CONDITION is a generalized instruction of considerable utility.

A more specialized conditional branch instruction is BRANCH ON COUNT, which reduces the value in a specified register by one, and which takes a branch to a specified address if the result of the reduced quantity is non-zero.

Another specialized area of conditional branching comprises two instructions: BRANCH ON INDEX HIGH and BRANCH ON INDEX LOW OR EQUAL. The functions of these instructions are essentially the same, except that the condition for branching is complementary. In execution, the value of an index is incremented and the new index quantity is then compared with a comparand. All values are contained in the general purpose registers. The structure permits the use of a single register for increment and comparand to give the equivalent of the 7090 TIX instruction. The branch is effected if the incremented register contents, when compared with the limit, take on the condition indicated in the instruction name. This instruction pair is carefully defined to be useful in the implementation of the FORTRAN DO statements. In general, the BRANCH ON INDEX LOW OR EQUAL would be used to close a loop at the bottom, and the BRANCH ON INDEX HIGH to close the loop from the top, the latter case permitting the extraction of vacuous cases. These instructions are defined so that they can remain invariant within executed code. All quantities that could possibly vary during execution (e.g., the index, the limit, and the increment, in that order of likelihood) are provided from registers. Only the specification of the general purpose registers containing these quantities, and the branch address itself, are included in the written instruction. These simple tasks, and indeed more general tasks, may be readily accomplished by the use of three instructions: a subtract or an add, a compare, and a BRANCH ON CONDITION. However, the simple tasks occur with a frequency sufficiently high to justify a form more easily specified and more rapidly executable.

subroutine linkage

Subroutine linkage is provided by the BRANCH AND LINK instruction. After the properly updated value of the program counter is stored in a specified general purpose register, control is transferred to the new address specified. Certain computer status information is stored together with the updated value. The additional information, placed in the leftmost eight bits of the register, consists of the length of the BRANCH AND LINK instruction utilized, the current state of the condition register, and the program mask (four bits defining the optional arithmetic conditions for which interrupts should occur). Except for the length, which may be of marginal value, the remaining information is significant for subroutine utilization. Subroutines frequently perform some function for the user and, upon completing this function, restore control to the user's sequence with the previous state of the condition register undisturbed. In another situation,

the subroutine may require a different program mask to properly monitor its execution, but must restore the user's program mask on return of control to the user's sequence. Both restorations are simultaneously accomplished by use of the instruction SET PROGRAM MASK, which loads the condition register and the program mask register from the leftmost eight bits of the specified general purpose register.

indirect execution

An EXECUTE instruction permits execution of an instruction that is not residing in the current instruction sequence. Except for an EXECUTE instruction, any instruction may be so executed. EXECUTE introduces an additional function to assist in attaining the objective of code invariance during program execution. This function permits modification of a portion of the instruction. The second byte of the operation code syllable may be modified by *or*-ing the low-order byte position of a general purpose register (specified by the first register address tag of the EXECUTE instruction) into the second byte of the operation code of the instruction to be executed as it resides in the operation decoding register. For example, this technique may provide the lengths of fields in variable-field-length instructions at execution time without program modification.

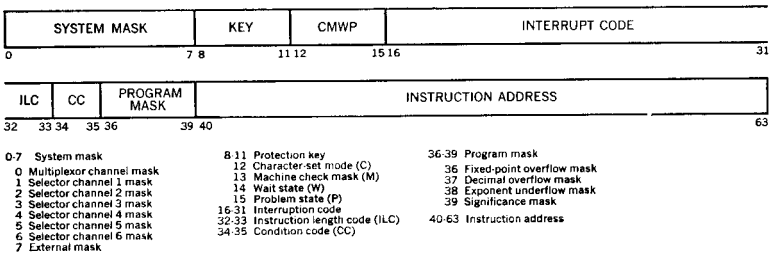
To permit greater flexibility in the selection of the instruction to be executed, double indexing is provided.

interruption

Only the basic structure, sequence of operation, and underlying philosophy of the SYSTEM/360 interruption facilities are described here. One of several broad considerations is that a constant hardware monitoring of machine status should have the capability of redirecting the CPU activity when prescribed situations arise. Another consideration is that the CPU must save adequate status information at the time of redirection to be able to return to an interrupted task. A primary consideration was that hardware provisions should perform all functions that must necessarily be automatic, such as storing volatile information, and those functions that are difficult or excessively time consuming to obtain by programmed operations. Any additional functions were considered a program responsibility.

The information collected and stored is called a program status word (PSW). As shown in Figure 2, a PSW is a double word, with the left word containing a system mask (defining the pro-

Figure 2 Program status word



hibited I/O and external signal interrupts), the storage protection key, the machine check mask and CPU mode bits, and a detailed specification of the cause of interrupt. The right word contains the same information that was stored in a general purpose register on execution of the BRANCH AND LINK instruction described earlier.

On occurrence of an interrupt, the PSW is stored in a double-word location prescribed for the particular class of interrupts to which it belongs. This is shown in Figure 3. Upon storing this old PSW, the CPU loads a new PSW from a second prescribed double-word location for this class of interrupts. The loading of this new PSW places the CPU into the appropriate state and directs the appropriate routine to properly service this class of interrupts.

The interrupts are classified by five types:

- Machine error.
- External signal.
- Input/output.
- Program error.
- Supervisor call.

The first two types are maskable, each by a single bit. The third type is maskable by individual I/O channel. The fourth type has a subset which is maskable by the program mask described with BRANCH AND LINK. The program masking permits election of automatic monitoring of any set of four arithmetic conditions: (1) floating-point underflow, (2) floating-point lost significance, (3) fixed-point binary overflow, and (4) decimal overflow. These conditions do not cause an interrupt unless indicated in the mask. The fifth type of interrupt need not be maskable.

Figure 3 Permanent storage assignment, interrupt area

	DOUBLE WORD (64)	
0	INIT PROG LOAD	PSW
1	INIT PROG LOAD	CCW 1
2	INIT PROG LOAD	CCW 2
3	EXTERNAL	OLD PSW
4	SUPERVISOR CALL	OLD PSW
5	PROGRAM	OLD PSW
6	MACHINE CHECK	OLD PSW
7	INPUT/OUTPUT	OLD PSW
8	CHANNEL STATUS WORD	
9	CHANNEL ADDRESS WORD	UNUSED
10	TIMER	UNUSED
11	EXTERNAL	NEW PSW
12	SUPERVISOR CALL	NEW PSW
13	PROGRAM	NEW PSW
14	MACHINE CHECK	NEW PSW
15	INPUT/OUTPUT	NEW PSW

Classification and selective masking readily permit the control of priorities between competing interrupt requests.

Monitor control

As computational speeds increase, relative to human reaction times, and as computing systems adapt to on-line and real-time multiprogramming tasks, the need for fully automatic systems increases. For such a fully automatic system, it is imperative that the allocation of system resources be controlled by a monitor program. Provision for this control is embodied in the following concepts:

- Monitor mode with associated privileged instructions.
- Storage protection to ensure the monitor's survival.
- Hardware monitoring, as described earlier, in conjunction with the ability to perform interrupts.
- An interval timer to periodically return control to the monitor.
- A wait state available to the monitor, rather than a stop or halt instruction available to the problem programmer.

storage protection

The primary function of storage protection is to prevent currently operating CPU or I/O channel programs from intruding into latent program and associated data areas. A number of different areas should be accessible and distinguishable from each other, because the I/O channel operations may be related to latent CPU programs rather than the current CPU program. In addition to providing separate identification of areas for different operating programs, it would be advantageous for storage allocation purposes to permit identification of several separated areas for one program.

To attain as full utilization of the storage capacity as possible, it is desirable to allocate storage in a piecemeal fashion. The allocation of storage is assumed to take place dynamically in an environment in which several programs occupy storage concurrently, new programs are being introduced into storage, and old programs are releasing storage. It is further assumed that a given active program may request or release additional data and/or program areas as its execution progresses.

In such an environment, contiguous storage space for a given program and its associated data areas cannot be long maintained without moving massive segments of information from one storage area to another. Such time-consuming movements of information can be reduced by permitting portions of storage to remain unused for periods of time, or by storage protection techniques that permit the allocation of the unused regions scattered throughout the storage of the computer. Simulation of such an environment indicates that unusable storage, in a system requiring contiguous allocation, ranges between 10 and 20 percent, depending on the distribution of storage requirements among jobs to be executed.

The storage protection technique selected for SYSTEM/360 permits arbitrary assignment of any number of 2,048-byte blocks of

storage to each of up to 16 active and latent programs. Storage protection is realized by providing each of the 2,048-byte blocks of storage with a 4-bit register. The monitor may store any 4-bit combination into any one of these registers. These combinations may be thought of as locks, each block of storage having its assigned lock. The 16 combinations are divided into two classes, zero and non-zero. The zero combination is considered as unlocked (either unassigned or common), and the 15 non-zero combinations are considered as locked.

The communicants with storage (namely the CPU, each selector channel, and the multiplex subchannel) are provided with independent 4-bit key combinations by the monitor. These key assignments are divided into two classes, zero and non-zero. The zero key, considered as the master key to all locks, is assigned only to appropriate sections of the monitor. The non-zero keys are considered as keys to matching locks only.

The protection function applies only to operations that store into a locked block. Storage takes place only if the key and lock combinations match or if the master key is used. Otherwise, the store is inhibited, and a program error interruption occurs.

Store protection alone suffices to provide shared programs in an environment of protected storage. The protection structure could be extended to include protection on fetches for privacy or vagrant reference monitoring. The extension could permit user options on the class of protection desired without compromising the options of other concurrent users.

I/O control

Although I/O operation is not in itself a monitor concept, the monitor program is inextricably involved in the division of I/O operating functions. To control the allocation of system resources, it is necessary that the monitor control the assignment and selection of all devices. Once the devices have been selected for use by a problem programmer, it is desirable that they be utilized as efficiently as possible. Since this utilization depends on the characteristics of the user's program, provision for controlling the desired sequence of operations is made available to the user.

The monitor program carries out the I/O device selection to the point where availability of the path to the device and of the device itself is determined. It then provides the I/O channel with the assigned storage protection key and the location of the first I/O command word to be executed.

The problem program supplies the desired sequence of commands to be carried out by the I/O device and the channel in concert. Any sequence of commands may be supplied, but all refer only to the one device selected. Both data chaining within physical records and command chaining between physical records may be performed. Termination of a chain of commands causes an interrupt status in the channel; for normal ending conditions, this interrupt can be suppressed.

Provision is also made to permit a channel-program-controlled interrupt by means of setting a flag bit in the appropriate com-

mand. Upon reaching this command, the channel takes on an interrupt status, but still continues executing the command(s). By means of this facility, the CPU can keep track of the progress of channel command execution. It is thus possible to dynamically allocate data areas as needed for records of unpredictable length.

interval timer

An interval timer is provided in byte locations 80 through 83 in storage. In the standard option, the contents of this word is decremented by 5 × 256 every sixtieth of a second. It is planned to provide an alternate decrementing by 6 × 256 every fiftieth of a second when 50-cycle-per-second power is utilized. By means of a higher-frequency oscillator, the multiplication by 256 permits greater resolution of the time interval to a minimum interval of about 13 microseconds. As the contents of this word location is decremented, the decremented value is tested: if the value has been reduced through zero, an external signal interrupt condition is initiated. This condition is honored by the CPU as soon as permitted by the CPU's state.

wait state

Maintaining control of system resources requires that no program can arbitrarily stop the CPU operation. When there is no call for activation of CPU programs, it may be desirable to place the CPU in a quiescent but responsive state. This is provided in SYSTEM/360 by a bit in the PSW, indicating the CPU to be in a "wait state." During this period, the CPU is quiescent, making no memory references; but it is responsive to any possible interrupt conditions, permitting immediate attention when called upon.

monitor mode

The CPU can be in either the monitor mode or in the problem program mode, as defined by a bit in the PSW currently controlling the CPU. In the monitor mode, all machine instructions may be executed. In the problem program mode, however, the class of instructions termed "privileged" cannot be executed, since it causes an interruption. The set of privileged instructions may be classified into three groups:

- Program status word (PSW) protection
  LOAD PSW
  SET SYSTEM MASK
- I/O facilities protection
  START I/O
  STOP I/O
  TEST I/O
  TEST CHANNEL
- Storage protection
  LOAD PROTECTION KEY
  STORE PROTECTION KEY

By virtue of storage protection in combination with the monitor mode and the associated privileged instructions, the monitor can maintain absolute control. No other program can accidentally or deliberately seize control without cooperation of the monitor program.

The monitor program protects the interrupt locations and itself, including all of its data used for resource allocation. Assignment of protection combinations to storage (STORE PROTECTION KEY) can be done only in the monitor mode. Assignment of the CPU protection key and the monitor mode state can be done in either the monitor mode (LOAD PSW) or by automatic reloading of a new PSW from monitor-controlled interrupt locations. These two provisions ensure the survival of all monitor functions. Any attempt to usurp these functions brings the offending program to the attention of the monitor.

Arithmetic functions

Even though the role of arithmetic functions in computing is the best understood functional class, many subtleties must be taken into consideration in an optimal choice of instruction formats, radices, representations, and word sizes.

Most data reduction, engineering, and scientific applications can accommodate themselves to suitable fixed word sizes. The performance payoffs of fixed word sizes, along with the ability to control accuracy by problem formulation, make this accommodation possible and somewhat attractive. Commercial data processing, on the other hand, deals with a variety of different but quite well specified field lengths; problem reformulation cannot materially alter the required or useful data lengths.

accumulators and instructions

An analysis of a representative sample of executed 7090 codes revealed that about 30 percent of the fetch and store functions were redundant. Data words were refetched whenever reused, and results were temporarily stored for later reuse. The analysis also demonstrated that significant reductions in redundancy could be achieved by providing multiple accumulators. The redundancy dropped quite sharply with the first few accumulators and tailed off thereafter. Provision of four accumulators accounted for more than 90 percent of the reducible redundancies, the fourth accumulator contributing only about 10 percent of this reduction.

Access time to storage increasingly limits performance as processor speed improves. Multiple fast-access accumulators permit a conceptual machine structure that depends less on storage technology by providing faster access to some of the data. Short register addresses, rather than larger storage addresses, reduce the instruction information required. The number of storage accesses needed to perform a given computational task is clearly reduced by multiple accumulators.

To effectively utilize multiple accumulators, two or more address fields are required in an instruction. Averaged over representative programs, two-address instructions are somewhat more efficient than three. To maximize this efficiency, the operations provided should be made available both between register pairs and between a register and a storage location. Lest the processor be coupled too closely to the speed of the storage unit,

results should normally go to a register rather than to a storage location. Stores should be separately provided.

indexing and fixed-point arithmetic

As indicated earlier, binary addressing was chosen because of efficiency of information storage and generality of table-lookup functions. It was also explained that by combining the fixed-point binary arithmetic unit with the indexing unit, the full power of the fixed-point arithmetic functions becomes directly available for address computations, and the results of fixed-point computations are directly available as index quantities in table lookup. These decisions circumvent a drawback of the conventional machine structure where fixed-point binary is associated with the floating-point unit. In very-high-performance scientific computers, floating-point operations are executed asynchronously and delayed with respect to address computations; as a result, the sharing of arithmetic facilities requires more complex interlocking than for separated facilities.

The binary representation chosen for the indexing and fixed-point numbers is 2's-complement. In this representation, the leftmost bit has a negative weight corresponding to its position, and all bits to its right have positive weights. Several consequences of this choice are: (1) multiple precision operations are simplified due to constant signs of low-order segments, (2) index quantities properly reflect the sign, even though the sign position is never involved in the effective address calculation, and (3) the truncation of a number always produces a "floor" (the largest integer less than or equal to the untruncated number) which is of considerable value in integer computations.

The normal fixed-point word size chosen is 32 bits, restricted to be on 32-bit boundaries to ensure maximum performance. This word size is sufficient to encompass the majority of fixed-point engineering or scientific calculations.

To improve storage efficiency and performance in data reduction applications, a 16-bit word is provided, which is sufficient to contain data from most sensing devices. These 16-bit words are extended toward the left to 32 bits as they are introduced into the arithmetic unit or accumulators. By virtue of this expansion, mixed 16-bit and 32-bit operands may be interacted arithmetically.

Multiple precision operations are assisted by several provisions: (1) logical addition and subtraction operations treat the leftmost bit of low-order segments as having positive weight, both for arithmetic results and for carries, (2) double-length dividends and products are accepted and generated, and (3) double-length register arithmetic and logical shifts are included.

Automatic monitoring of fixed-point overflow by the interrupt mechanism, provided as a programmer option, is one of four maskable interrupt conditions. Divide check is at all times monitored by the interrupt mechanism.

floating-point arithmetic

The choice of representation of floating-point numbers was to some extent tied to the choice of word sizes. A common exponent and sign representation was considered advantageous be-

cause it is needed in economical provisions for interacting operands of differing size. A 64-bit word size is provided for the high-precision requirements. Although the 36-bit word size had apparently been adequate for more than 95 percent of the 7090 applications, it was considered more marginal as problem sizes and complexities grew. However, an additional 32-bit word size is provided for the sizeable number of applications for which this precision is entirely adequate. Both word sizes are restricted to be on boundaries of their size, again to ensure maximum performance.

Precision greater than that of the 64-bit word is programmable. However, no special provision for this case was considered necessary because of the low frequency of multiple precision coding on the 7090. It is assumed that only a small fraction of this coding would not be adequately served, since the 64-bit representation chosen is equivalent to a double precision 7090 representation.

In the floating-point representation adopted, the leftmost 8-bit byte of either of the two word sizes contains the fraction's sign and the exponent (excess 64). The byte size was chosen to maintain consistency with data boundaries. The remaining three and seven bytes, respectively, of the two word sizes represent the absolute value of the fraction. The exponent represents a power of 16 rather than of 2, so the fraction is normalized in 4-bit shifts rather than single-bit shifts. The use of radix 16 permits twice the effective range of the exponent in the 7090 representation. An analysis of experimental computation indicates that, on the average, approximately 2.3 of the leftmost fraction bits are zeros.

When in the arithmetic section of the computer, a 32-bit floating-point word occupies the left half of a 64-bit floating-point register. Except for products, all results of loads or arithmetic operations leave the right half of these registers undisturbed. In the case of multiplication, the right half is replaced with the low-order part of the product, allowing multiplication and summation with minimum accumulation of truncation error. By not disturbing the right half of the accumulator contents, higher performance is achieved on computers with data flow widths of less than 64 bits.

To aid in retaining the maximum significance in the 32-bit addition and subtraction operations, a 4-bit digit of a preshifted operand is retained to the right of the word boundary when introduced into the floating-point adder. If a left shift of the resulting fraction is required to maintain normalization, this additional digit is retained in the result. No corresponding provision was felt necessary for the 64-bit word size.

In floating-point division, no remainder is retained. The use of the remainder in floating-point computation is far less frequent than in fixed point. Moreover, a large share of the uses encountered can be achieved by alternative operations of nearly equal efficiency. Retaining the remainder would require another accumulator for each division, reducing the effectiveness of the four accumulators during most floating-point computations. A similar argument

eliminates the use of an accumulator for holding the low-order segment of the product of 64-bit words.

Automatic monitoring of exponent underflow and of lost significance (vanishing fraction) by the interrupt mechanism are provided as separate programmer options. These are two of the four maskable interrupt conditions. Appearance of either condition causes the result to be set to zeros, a condition treated as a true zero by the computer. Division by zero and exponent overflow are at all times monitored by the interrupt mechanism.

decimal arithmetic

It was indicated earlier that two decimal digits are represented in one 8-bit character or byte. Decimal number fields, then, are an integral number of such bytes. The algebraic sign of the decimal number occupies the rightmost decimal digit position in this field, providing an odd number of decimal digits plus sign.

The maximum length of a decimal field must be adequate to permit representation of numbers that may be declared in COBOL, therefore requiring fields up to 18 decimal digits plus sign. The actual field size readily permitted by the structure is 31 decimal digits plus sign, providing a suitable margin for untruncated products occurring during computations.

Considerable effort was expended in determining whether registers or storage locations should be used for accumulators in decimal arithmetic operations. Sharing of the floating-point accumulator for this purpose would yield two decimal accumulators. Several small problems were coded and timed, yielding no particularly conclusive evidence. To aid in resolving this alternative, several typical COBOL source programs were analyzed. These programs showed clearly that the arithmetic strings were very short, and that normally a large number of accumulators would be referenced before succeeding strings would apply to the same accumulator contents. This information clearly suggested retention of accumulator fields in storage, thus eliminating many redundant store and load operations, and precluding strong pressures for global optimization of COBOL codes for most efficient register usage.

Operations on operands of differing size are permitted by the instruction format. Two field-length tags are associated with two corresponding storage addresses. The first tag specifies the length of the accumulator or result, and the second tag specifies the length of the interacting operand. Improper specifications, e.g., the absence of a sign on the right of the field or a sign embedded in a field, are recognized by the decimal arithmetic unit and initiate an interrupt. Automatic monitoring of accumulator overflow by the interrupt mechanism is a programmer option—another one of the four maskable interrupt conditions. Divide check is at all times monitored by the interrupt mechanism as in fixed-point arithmetic.

Non-arithmetic data manipulation

The non-arithmetic operations are still not well understood; no well-ordered formal procedures have been constructed for dealing

with the general problem of logical data manipulations. In spite of such limitations, these operations form a substantial portion of today's computing task. They normally appear in the programmed interface between human-oriented language and machine-oriented language. Their principal uses are in the extracting, categorizing, transforming, rearranging, and editing of information entering or leaving the computer system. The operations are discussed in groups that possess similar functions.

shifting

Shifting operations are used for many of the processes of isolating, concatenating, and aligning groups of contiguous bits.

Left and right shifts, specifying either a single register or an even-odd adjacent register pair, are provided. The amount of shift is specified by an address syllable, allowing specification by immediate or register information, or by both. Although the results after shifting are tested on the corresponding set of arithmetic shifts, testing after logical shifts is left for subsequent operations. A single set of such tests was found to be of marginal utility.

operand-pair logic

Logic operations on operand pairs are principally intended for extracting, testing, modifying, and recombining bit groups which, if non-contiguous, are localized in bytes, words, or byte strings.

Provision for these operations is variously made in four of the five instruction formats. The register-to-register operations are all between 32-bit words. The storage-to-register operations are also between 32-bit words, with the exception of the pair for single-character load and store. All storage-to-register operations are doubly indexed to enhance addressing flexibility. The storage-to-storage operations are performed between two equal-length operands, each ranging from 1 to 256 bytes. All immediate-to-storage operations are between the second byte of the operation code syllable of the instruction and the byte addressed in storage, the latter byte being in all cases the result byte for these operations.

The data moving instructions are described first. Four have the storage-to-register format: LOAD and STORE for 32-bit operands, and INSERT CHARACTER and STORE CHARACTER for 8-bit operands. INSERT CHARACTER alters the low-order byte of the specified register only, replacing it with the byte from storage. In the other three formats, a single MOVE instruction is provided. The data moving operations leave the condition register undisturbed. Although it would be desirable at times to make tests during the movement of data, it is more advantageous to be able to move data between a deliberate test operation and the conditional branch on this test.

A second group is formed by COMPARE LOGICAL, which appears in four formats and treats all bits within the operand boundaries as having positional binary magnitude properties. The condition register is set to one of three states: low, equal, or high.

A third group consists of AND, OR, and EXCLUSIVE OR, which appear in all formats and consider all bits within the operand boundaries as independent of each other. AND is normally used to force 0's into certain bit positions, OR to force 1's, and

EXCLUSIVE OR to alternate the binary values of these bit positions. These operations set the condition register to either the zero or the non-zero state, depending upon the magnitude of the result.

A final group consists of the one instruction TEST UNDER MASK, which appears in the immediate-to-storage format only. TEST UNDER MASK does not alter the byte in storage, but selects those bits from the byte in storage that are specified by 1's in the corresponding bit positions of the immediate byte. This collection of selected bits is tested for three conditions: all 0's, all 1's, or mixed 0's and 1's. The condition register is set in accordance with the outcome.

Code translation and recognition functions are provided by the instructions TRANSLATE and TRANSLATE AND TEST. Both instructions provide two storage addresses. The first address specifies an argument operand, ranging from 1 to 256 bytes in length. The second address specifies the origin of a function table of bytes defining the desired translation or testing transformation.

The TRANSLATE instruction performs a history-independent translation. Each byte of the argument, scanning from left to right, is in turn replaced by a function byte from the table. The location of the function byte is obtained by adding the argument byte to the table origin. The condition register is not altered as a result of this operation.

TRANSLATE is useful for rearranging records as well as for code translation. For this usage, the record to be rearranged is considered the function table, and the desired rearrangement pattern is given by the argument. Each argument byte specifies a location in the function record. In the process of translation, the argument bytes are replaced by the appropriate bytes from the function.

The name of the instruction TRANSLATE AND TEST is somewhat a misnomer. Up to the point of replacing the argument byte with the function byte, the execution is similar to that of TRANSLATE. However, no actual translation takes place. Instead, each function byte is inspected to see if it is zero or non-zero. If the function byte is zero, the corresponding argument byte is considered of no interest, and the process repeats on the succeeding argument byte. A non-zero byte, however, is considered of interest, and three actions are taken: (1) the address of the argument byte is inserted in the three low-order byte positions of register 1, (2) the non-zero function byte is inserted in the low-order byte position of register 2, and (3) the operation is terminated.

The address saved is generally useful for three purposes. Most obvious but least useful is the fetching of the argument byte of interest. In most cases, the significance of this byte is defined by the function byte retained, which may be used for branching to an appropriate routine. The address can specify the starting address for continuing a TRANSLATE AND TEST operation, and can be used to determine the terminal point of a sequence of un-

interesting argument bytes. Such a sequence constitutes an item of information (such as a name in a FORTRAN statement or a message in a communication from a console) if the interesting byte is a delimiter.

The condition register is set to one of three states: (1) all function bytes encountered are zero, (2) the argument is not yet fully translated or ended on a zero function byte, or (3) the final byte of the argument had a non-zero function byte.

editing

The editing functions described here are determined in part by the nature of the extended code employed in IBM cards and the practices developed during its history, and in part by the use of the 8-bit byte for external communication. Moreover, various editing conventions are met for user convenience. The editing instructions fall into three categories of two instructions each.

The first two instructions are based on the frequent use of zone overpunching of numeric fields on IBM cards to hold class information. These instructions, MOVE ZONES and MOVE NUMERICS, permit the separation or recombination of zone and numeric data. MOVE ZONES reads the four high-order bits of each byte from the source operand location and inserts them into the four high-order bits of the bytes in the sink operand, leaving the four low-order bits undisturbed. MOVE NUMERICS performs the equivalent operation on the four low-order bits. Both operands are of the same length, which may range from 1 to 256 bytes. No condition register setting is made.

PACK and UNPACK are used for packing and unpacking decimal fields on IBM cards; these instructions assume the probable use of sign overpunching on the low-order digit of the decimal field. The instructions specify two operand fields of independent lengths, each ranging from 1 to 16 bytes. The operands are processed from right to left.

PACK takes the first byte of the source, switches zone and numeric positions, and stores the result into the first byte position of the sink, thus placing the probable sign into the proper position. From this point on, the numeric fields of each pair of successive source bytes are compressed into one byte of the sink. If the source runs out before the sink field is filled, the sink is filled out with zeros. If the sink field fills before the source runs out, the operation is terminated. UNPACK is very nearly the inverse of PACK, except that the zones to be associated with the unpacked numerics are automatically supplied for all but the first byte, including any zeros used to fill out the unpacked sink field. Because neither of these operations is contingent upon the presence of sign overpunching, but merely accommodate this practice, they also permit rapid editing of hexadecimal information.

A third instruction pair permits editing of packed decimal information for printing. EDIT and EDIT AND MARK suppress or protect leading zeros, and can also provide punctuation (such as commas and periods) and suppression of sign-controlled print fields. EDIT AND MARK indicates the position for proper insertion

of a dollar sign. Each of these instructions specifies two operand fields. The source is a packed decimal field or a series of packed decimal fields. The sink is an editing pattern, and the length specification applies to the pattern field. An operation proceeds from left to right and is completed in one pass.

In execution, use is made of a significance trigger which starts in the *off* position. The trigger is turned *on* either when the first non-zero source digit encountered is requested by a digit-select character in the editing pattern, or when a significance-start character is encountered in the editing pattern. The trigger is turned *off* again either upon encountering a positive sign code as the next source digit (immediately, with no waiting for a digit-select character in the editing pattern) or upon encountering a field-separator character in the editing pattern.

A pattern character is first inspected to determine the function to be performed. After performance of this function, the pattern character is replaced by a fill character if the significance trigger is *off*. This permits the suppression of significance-dependent characters, such as punctuation, and sign-dependent characters, e.g., credit symbols. The fill character employed is the first character encountered in the editing pattern. If this character is also a functional character, the function is performed.

If the significance trigger is *on* after the function requested by the pattern character has been performed, any one of three operations may be carried out: (1) if the pattern contains a digit-select or a significance-start character, the current source digit is expanded to zoned format and stored over the pattern character; (2) if the pattern contains a field-separator character, it is replaced by the fill character; (3) if the pattern contains any other character, it is left undisturbed.

The source field is advanced to the succeeding digit position either after encountering a digit-select or significance-start character in the editing pattern, or after encountering a sign code as a source digit.

In EDIT AND MARK, an additional operation is performed. Whenever the significance trigger is first turned *on* by the appearance of a non-zero digit, the address of this byte position in the editing pattern is inserted into the three low-order bytes of register 1. After completion of editing, the condition register is set to the sign of the last field edited.

conversion

Conversion operations are provided for radix conversion of address and data values. The CONVERT TO BINARY and CONVERT TO DECIMAL instructions appear in the storage-to-register format with double indexing. The binary operand always appears in the register and is 32 bits long. The decimal operand always appears in storage in a 64-bit field aligned on double word boundaries. Both operands are treated as signed integers, the binary operand having the 2's complement form when negative, and the decimal operand having a packed-decimal magnitude with the low-order digit position containing the sign code.

Overflow can occur on execution of CONVERT TO BINARY, because the result can overflow the register capacity. In this event, the 32 low-order bits of the appropriately converted number are retained in the register, and an interrupt is taken.

Summary

The design objectives laid down for SYSTEM/360 included requirements for large memory capacities, simple program relocation, flexible storage protection, and general supervisory facilities. Also required were a variety of data formats, an extensive set of processing operations, and machine-language compatibility among models of widely varying performance.

This paper comments upon the addressing, sequencing, monitor control, and central processing solutions that were adopted to reconcile these requirements and to meet the needs of diverse computer applications.