gri-909

THE NEW BREED

system reference manual

gri GRI Computer Corporation

76 ROWE STREET, NEWTON, MASSACHUSETTS 02166

Attached are correction sheets for the GRI-909 System Reference Manual, Rev. A, December 1969.

GRI Computer Corporation
April 10, 1970

Price $3.50

gri-909*

System Reference Manual

December 1969

GRI Computer Corporation, 76 Rowe Street, Newton, Massachusetts 02166

*Patents Pending

The information given herein is for reference purposes only and is not to be taken as engineering specifications.

Printed in USA

CONTENTS

CHAPTER ONE

INTRODUCTION

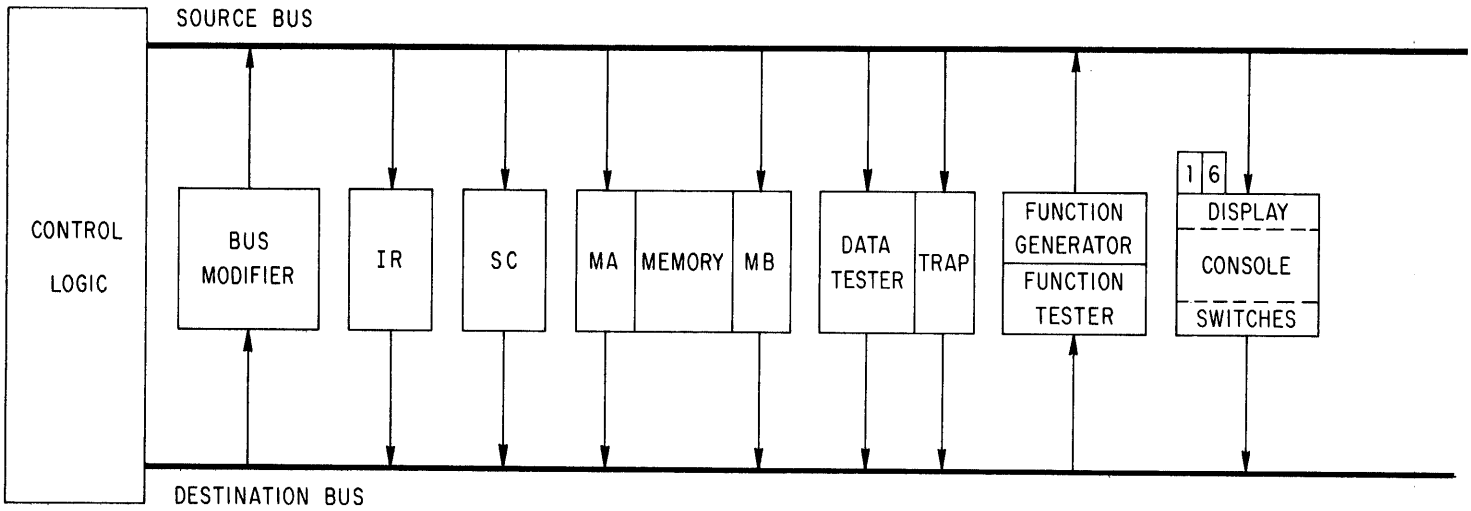
The GRI-909 is an advanced general purpose computer, whose highly modular functional design and easy, efficient programming make it especially suitable for applications in process or system control. The machine is based on the concept of Direct Function Processing and is organized around two buses, a source bus and a destination bus. As shown in the illustration on the next two pages, all internal control registers, device controls, arithmetic units, function generators, and even memory itself (all referred to as "operators") exist between these buses. Each bus comprises sixteen data lines, six address lines for operator selection, control lines, sense lines, and lines for timing signals. A typical operator accepts data and/or control signals from the source bus, acts according to the received control signals and the nature of the operator, and outputs its result onto the destination bus. The bus modifier provides a path for moving data from the destination bus to the source bus, where it is available as input to the next specified operator. The function tester senses control signals from other operators via the destination bus; the function generator supplies control signals to other operators via the source bus. This structure provides many intrinsic advantages over other computer architectures. Among these are:

The ability to program in an assembly language directly related to system functions without the inefficiencies in time and memory introduced by compilers.

The capability of direct communication between system devices and computer components. Any system device can directly access memory, and the processor can transfer data between any two system devices without tying up accumulators or special registers. This eliminates the "bookkeeping" instructions otherwise needed to manipulate data to, from and through the processor.

The ability to expand the instruction repertoire on a modular basis including many special or unique instructions.

The minimum processor configuration comprises all operators shown on the next page except memory, which is optional. The memory can contain both alterable core memory modules and read-only memory modules, and the two are interchangeable; if only the latter is present, the computer can be operated simply as a hardwired controller. Shown on page 1-3 are typical options including functional or "firmware" operators and device operators. Device operators are those that interface peripheral equipment such as input-output devices, mass memory media, communications equipment and displays. Functional operators are those that operate on data words or perform various functions that are usually regarded as internal to a computer: these include the real time clock, an arithmetic operator that performs addition and basic logical opera-



tions, and operators that perform multiplication, division, byte swapping and packing, and other more complex functions.

The basic package requires only 10½ inches mounted in a standard 19-inch rack. It contains the processor, a memory of up to 8K words, and has space for three major firmware options and sixteen other optional operators of either type. Additional memory can be held in an expansion chassis mounted above the basic package; an expansion chassis for other options can be mounted below it.

General Characteristics

16-bit parallel processing, with a 1.76 microsecond cycle time when executing instructions from main memory, an 880 nanosecond cycle time when executing instructions from an external memory, *ie* a read-only memory contained in a firmware option.

32,768 words of directly addressable, random access core memory. Minimum core size is 1024 words.

The functional organization permits programming in a simple system-oriented assembly language. Programs can be assembled on an IBM 360 computer.

Every device in the system, both inside and outside the processor, is directly addressable by programmed instructions, allowing direct data transfer between devices without need for special accumulators or temporary storage.

Firmware options can expand the hardwired instruction set to provide system flexibility unequaled by more conventional computer designs.

Direct memory access can be made via the standard source and destination buses at the rate of 1.76 microseconds per 16-bit word (568,000 words per second). No multiplexer is required.

Priority interrupts can be executed on a single level or on sixteen levels at the choice of the system designer. Additional interrupt levels are available as an option.

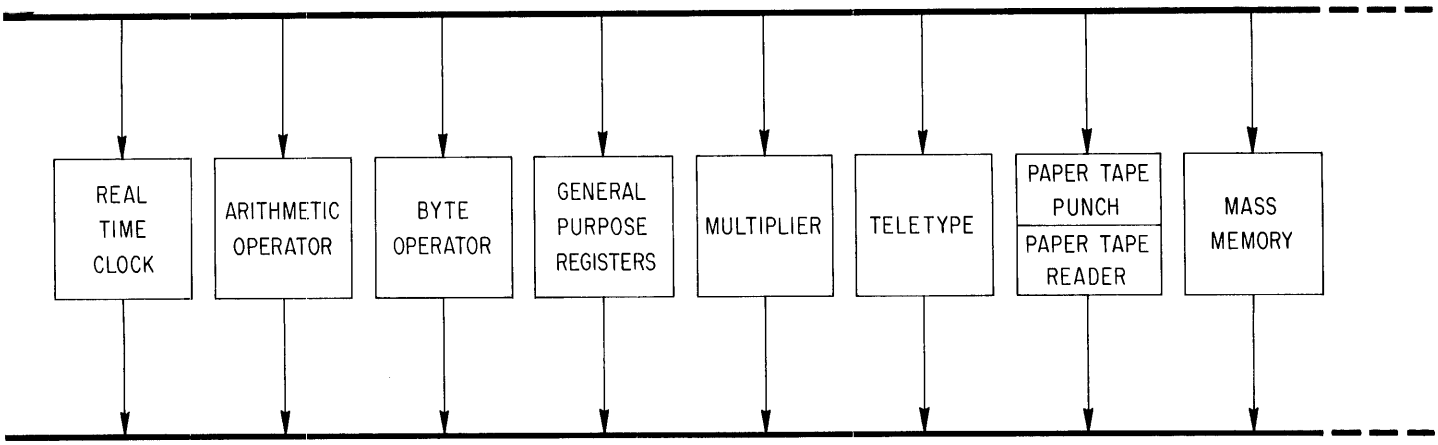
Core memory and read-only memory are interchangeable.

Power failure protection and automatic restart are standard. Remote start and stop are available for use in a system interface.

Any register, whether inside or outside the processor, can be displayed on the console. Data can be transferred from the console to any register.

Specifications

Physical. 10½ inches high, 20 inches



deep, weighs 50 pounds. Mounts from the front in a standard 19-inch rack with or without slides.

Electrical. Single phase line power, either 100-130 vac, 60 Hz ± 3%, or 200-240 vac, 50 Hz ± 3%. 4 amperes, 150-250 watts. Logic levels: ground and +4 vdc, DTL and TTL compatible.

Environmental. 0° to 50°C ambient temperature. Relative humidity to 90%. Cooling is by convection (no fans required for operation over the ambient temperature range). All console switches are photo-optical without mechanical contacts subject to wear or bounce.

1.1 COMPUTER ORGANIZATION

The direct function processor controls the entire system by controlling all functional and device operators. The processor handles words of sixteen bits, which are stored in a memory with a maximum capacity of 32,768 words. The bits of a word are numbered 0 to 15, right to left, as are the bits in the registers that handle the words (each bit number corresponds to the magnitude of the bit as a power of 2). Memory addresses are fifteen bits, numbered according to the position of the address in a word, *ie* 0 to 14. Words are used either as computer instructions in a program, as addresses, or as operands (data for the program). The program can interpret an operand as a logical word, an address, a pair of 8-bit bytes, or a 16-digit signed or unsigned binary number. Arithmetic operations are performed on fixed point binary numbers, either unsigned or the equivalent signed numbers using twos complement conventions.

The processor performs a program by executing instructions retrieved from memory locations addressed by the low order fifteen bits of the sequence counter, SC. At the end of each instruction that does not reference memory, SC is incremented by one so that the next instruction is normally taken from the next consecutive location. In an instruction that does reference memory, SC is incremented once so that the instruction can make use of the next location — for either an address or an operand — and it is then incremented again to point to the next instruction. Sequential program flow is altered by changing the contents of SC, either by incrementing it an extra time in a function test instruction, or by replacing its contents with the value specified by a jump instruction. In a jump, an address for returning to the original sequence is saved in the trap register, TRP, which is also available for temporary storage. Since SC is connected across the buses, either the program or an external operator

can alter the sequence by transmitting an address directly to SC. The memory address register, MA, supplies the address for every memory access, and words are transmitted between the buses and the memory via the memory buffer, MB. When each instruction in the stored program is taken from memory, it is sent to the instruction register, IR, for decoding and execution by the processor.

The path for data controlled by the program is from a source operator to the destination bus, through the bus modifier to the source bus, and then to a destination operator. For its input data, the bus modifier can take either the source word or its complement and can modify it in transmission in one of the following ways: increment it by one, shift it left one bit, or shift it right one bit. Thus the twos complement negative of a number can be obtained during transmission by combining the complement and increment operations. Associated with these functions are an overflow flag and a link bit that connects the ends of the word for circular shifting.

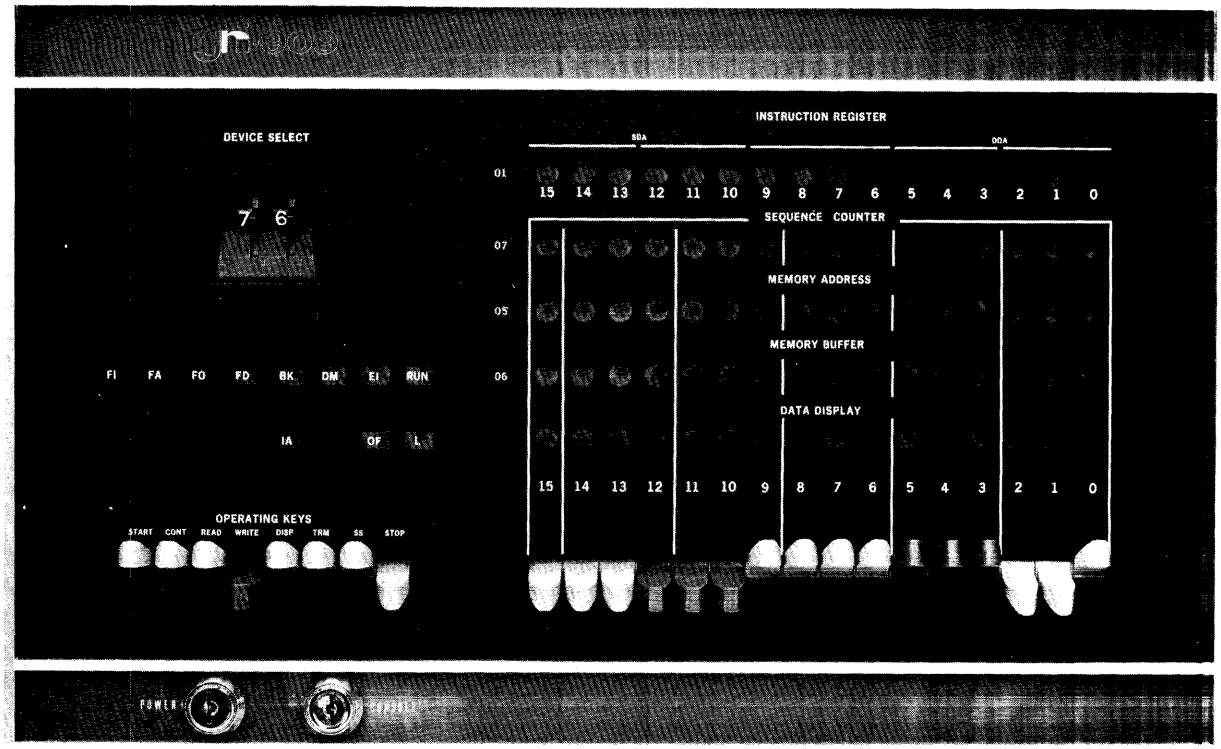
The other three programmable functional operators that are basic to the organization of the system allow the computer to perform tests on data and functions, and to generate functions. The data tester determines whether the arithmetic value of a word it receives is less than, equal to or greater than zero, or any meaningful combination thereof. If the test result satisfies the condition specified by the program, the current contents of SC are stored in TRP and a jump is executed.

Through the function generator a single instruction can send up to sixteen independent control functions to any operator. In this way the function generating instructions control the peripheral equipment and trigger the functional operations necessary for effective programmed use of the system. By means of the function tester a single instruction can sense up to three control signals supplied by any operator; *eg* these instructions sense the status of peripheral devices. A function testing instruction can select one or more of the control signals and can test whether any of the selected signals is true or if all those selected are false; a positive test result produces a skip.

Two plug-in, interchangeable consoles are available for the processor. The programmer's console has lights for simultaneous display of IR, SC, MA, MB, and any selected data. The usual procedure is to use the programmer's console while debugging system software, and then substitute the basic model when the system is installed. Any register, either internal or external to the computer, can be selected for display on either console by a pair of thumbwheel switches. The program can read the contents of the switch register at any time, and a word sent to the destination dialed into the thumbwheels is automatically displayed in the lights. This permits display and debugging of all system devices directly from the computer console.

The system described thus far is the minimum configuration in which the GRI-909 is available. With the addition of some kind of stored program, it can provide all the control capabilities of a general purpose computer, although arithmetic and logical operations must be performed one bit at a time using the bus modifier. Random access core memory can be added to the system in units of 1024 and 4096 words to a maximum of 32,768. With core memory the processor also has facilities for program interrupts and high speed data transfers.

The interrupt system facilitates processor control of peripheral equipment by allowing any device to interrupt the normal program flow on a priority basis. The processor acknowledges an interrupt request by storing SC in a memory location and executing the instruction contained in the next consecutive location. The system can be set up so that a unique triplet of locations is assigned to each device, or all devices can interrupt to location 0, and the program must determine which device requires service.



Console

A high speed device, such as magnetic tape or disk, can gain direct access to memory without requiring the execution of any instructions; the program simply pauses for a single cycle while access is made. The logic for direct memory access allows the transfer of data to or from memory, or the incrementing of a memory word, a feature useful in such applications as high speed pulse counting.

This minimal version of the GRI-909 provides an economic controller for applications with limited arithmetic requirements or where arithmetic execution time is not a factor. Such applications might be communication traffic control, data concentration and formatting, peripheral equipment control, and simple system control. The addition of plug-in hardwired or read-only memory instructions allows the GRI-909 to be expanded to encompass many powerful features on a modular basis.

The most basic firmware option that can be added to the system is an arithmetic operator. This operator contains two registers and a functional data output whose generation does not disturb the contents of the registers. The program can produce arithmetic and logical functions of the contents of these registers by placing the arithmetic operator in the appropriate functional state. The functions that can be performed are addition and the three logic functions, AND, inclusive OR, and exclusive OR. At any given time the data output of the arithmetic operator is equal to the selected function of the contents of the two registers. The output changes automatically by changing the functional state of the operator or the contents of either register. Hence the program can compare a series of values against some limit stored in one register without disturbing that register or requiring temporary storage.

Other firmware options include general purpose registers and operators that swap the bytes in a word, that pack bytes into a word, and that perform computations

such as multiplication, division, square root, and so forth. In many cases the user has a choice of options for performing a given operation at various speeds. Multiplication can be done by a subroutine with or without an arithmetic operator, but two multiply options are also available. One performs a multiplication in about 56 microseconds by requiring the processor to perform external instructions, *ie* instructions supplied by a read-only memory in the operator; the other is fully implemented by internal hardware and takes under 10 microseconds.

The execution of external instructions is a technique that can also be used by hardwired diagnostics, supplied in the form of plug-in modules, that can exercise all nonmemory registers in the system.

1.2 MEMORY

From an addressing point of view, the entire memory is a set of contiguous locations whose addresses range from zero to a maximum dependent upon the capacity of the particular installation. In a system with the greatest possible capacity, the largest address is octal 77777, decimal 32,767. Core memory is available in increments of 1024 words or 4096 words. A given system can contain at most four of the 1K modules; a typical system might contain 4096 words of core memory and 1024 words of read-only memory. Read-only memory differs from core only in that it cannot be altered by the program. Common software routines are available in standard read-only modules, others are available on a custom basis. (Some firmware options actually contain their own small read-only memories, but these are not addressable by the program.)

Memory Allocation. The only locations whose use is fixed by the hardware are those allocated to the program interrupt system and autorestart, which uses locations 6 and 7. System operators can be wired to interrupt to any location, but the standard product line operators are wired to use locations 0-5 and 11-62. Moreover the user can assign all devices to location 0, in which case an interrupt causes the processor to save SC in location 0 and execute the instruction in location 1. Any location used by the interrupt or autorestart should ordinarily be regarded as unavailable for general programming.

1.3 INSTRUCTIONS

Every instruction can address two operators, one as the source of information, the other as the destination. These instructions are of the general form, *Operator X to Operator Y*. Every operator in the system, from internal control registers to peripheral devices, can be addressed with the one exception of the bus modifier, which plays its role automatically. Thus instead of performing a jump, the program can alter its own sequence simply by transmitting a word to SC. Generally IR and MA are addressed only by the processor for its own internal functions; the program can read these registers but cannot load them since such action would be meaningless. In other respects there is no intrinsic difference between the automatic operations performed by the processor and the operations specified by the program. In reality the processor executes a built-in sequence of instructions into which the instructions from the program are inserted.

Consider an instruction for a simple transfer from one nonmemory register to another. As an instruction in the program it is executed in a single processor cycle; but the cycle is actually divided into four parts in which the processor executes four instructions. The first step is a transfer from SC to MA to retrieve the program instruction from memory; when it is available it is transferred from memory to IR. In the third step the processor decodes the instruction in IR

and executes it. In the fourth and final step SC is moved to itself via the bus modifier but it is incremented by one along the way.

The actual type of operation performed and the type of information moved by any instruction depends upon the operators it addresses. For a simple transfer the programmer can take either the word directly from the source operator or its complement, and he can elect to modify the word as it passes through the bus modifier. An instruction that specifies memory as source or destination automatically uses the location following itself in memory. Besides selecting the type of modification (if any) of the word in transit, the instruction also selects the addressing mode, *ie* whether the next memory location is used to receive or supply an operand, or to supply an address to be used in locating an operand.

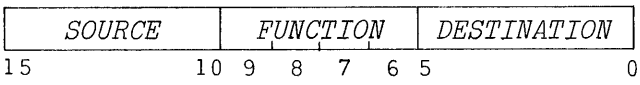
Other types of instructions are produced by addressing appropriate operators either as source or destination. The program tests a source word for a jump by addressing the data tester as its destination. The program generates control signals for a destination operator by addressing the function generator as source, and tests control signals from a source by specifying the function tester as destination.

Using these basic types of instructions the program controls the remaining operators in the system. The two registers in the arithmetic operator are addressable for data transmission purposes; by sending control signals from the function generator, the program can specify the arithmetic or logical operation the unit is to perform, and the result is available for data transmission. Sending a word to the byte swapper makes the word, with its bytes swapped, retrievable by addressing the byte swapper as source. Even though all instructions are of the same general form — there are no special in-out instructions — nonetheless, control over a peripheral device is essentially like that in any computer: data transmission instructions send data to and from the interface, and the function generating and function testing instructions control it. The addition of any new peripheral device or firmware option increases the instruction set in all types for which it is meaningful to address the new operator. Generally it will require function generating and testing instructions to control it and sense its status. If it is a source of data, its output can be sent, with or without modification, to any data destination including memory: conversely, if it can receive data, it can be addressed as destination in a transmission instruction that sends data (perhaps modified) to it from any source throughout the entire system.

There are several levels at which operations can be performed. With only the basic processor, multiplication must be performed by a software subroutine. If general purpose registers are added, the subroutine is shorter; adding an arithmetic operator shortens it more; and with both it is shorter yet. But multiplication can also be performed directly by an optional operator added to the system. In some cases a function is implemented entirely by hardware: the program sets up the operator and gives an instruction that causes it to perform its operations internally. The arithmetic operator is of this type, and a hardware multiplier is also available. But there is another multiplication operator that must be set up by the program, and the program then gives an instruction to place it in operation, but the operator does not perform all of its operations internally. Instead it takes over the processor temporarily and causes the processor to execute external instructions retrieved from a read-only memory built into the multiplier. Of course the operator contains enough hardware so that the external routine is much shorter than a software routine that uses the arithmetic unit and general purpose registers. But as far as the program is concerned, and as far as other operators in the system are concerned, the entire sequence behaves like a single instruction. In other words the instruction that says "go", plus the ex-

ternal subroutine, looks to the programmer like a longer instruction that says "multiply". External instructions cannot reference memory, and they are executed at twice the rate required for instructions stored in main memory.

Instruction Format. There is one basic format for all instructions in the GRI-909. In the 16-bit instruction word, the left and right six bits respectively (bits 10-15 and 0-5) are the codes of the operators that are the source and destination of the information transfer, and the middle four bits are used for control or functional information.



If the elements named as source and destination are simply registers that can supply or receive data and are not in memory, then the instruction is simply for data transmission and the function bits specify the way in which the word taken from the source is modified before being sent to the destination. In some cases the actual transmission may cause the destination to perform some function; *eg* sending a character to the tape punch causes that device to punch the character, but the instruction is still classed as data transmission. If the instruction is for data transmission, but the operator specified as source or destination is memory, the contents of the next consecutive location following the instruction are used either as the data word transmitted or as an address to locate that word. The function bits in the instruction specify not only the modification of the word in transit, but how the word following the instruction is to be used.

Other types of instructions are produced by naming as source or destination an operator that is not simply a register to hold data. Naming the function generator as source supplies up to four control signals to the named destination, such signals being selected by the function bits of the instruction. Specifying the function tester as destination causes it to test signals from the source for a skip: bits 6-9 select the signals and the conditions they must satisfy. Naming the data tester as destination causes it to test a word supplied by the source for the conditions specified by the function bits; if the condition holds, the processor jumps to a location determined from the word following the instruction in memory.

Thus even though all instructions are of the same form, the repertoire includes a variety of operation types depending upon the properties of the operators addressed. These run the gamut from basic core memory storage to peripheral devices such as line printer and magnetic tape, to functional operators for performing arithmetic calculations, to purely control devices that generate or test functions.

Effective Address. An instruction that specifies memory as source or destination or specifies the data tester as destination takes two words, *ie* it uses the next consecutive memory location following the instruction. Data may be stored in the second location, or its contents may be used either as data or as an address.

For any instruction of these types the processor must determine the effective address, which is the actual memory address used to fetch or store the operand or alter program flow. A data transmission instruction that references memory can specify that addressing is direct, deferred (also called indirect), immediate, or immediate and deferred. A jump instruction, *ie* one that names the data tester as destination, can specify only direct or deferred addressing. With direct addressing bits 14-0 of the location following the instruction are used as the effective address, *ie* the address of the location to be used for retrieval or storage of data or retrieval of the next instruction (of course the latter case holds in a conditional jump only if the condition is satisfied). For deferred addressing the processor retrieves a word from the location addressed by the contents of the

location following the instruction; it then adds one to that word, writes the result back in the same location and uses it as the effective address.

Immediate mode addressing can be used only by the data transmission instructions. In this case the effective address is simply one greater than the address of the instruction; in other words the contents of the location following the instruction are used as the operand or that location is the destination for an operand. If addressing is both immediate and deferred, the processor takes the word from the location following the instruction, increments it by one, places the incremented word back in the same location and uses it as the effective address.

Programming Conventions. Although all instructions have essentially the same format, the assembler distinguishes four classes: data transmission, data testing, function generating, and function testing. The assembler also distinguishes data transmission instructions that reference memory from those that do not.

The GRI-909 actually has two assembly programs, a basic assembler (BASE) and a functional system-oriented assembler (FAST), that use very different assembly languages. One is a terse, symbolic language like those used with most assemblers; the other is a compiler-like system language that is closer to ordinary English. The latter provides a much easier introduction to the computer, but the former is more effieient to use once one gains familiarity with the GRI-909. Instruction descriptions and program examples in the text are given in the basic assembly language. A detailed treatment of FAST is given in Appendix A, although Chapter 2 does give the general forms for the various instruction types in the system language.

The basic assembly program recognizes a number of mnemonics and other initial symbols that facilitate constructing complete instruction words and organizing them into a program [*Appendix E*]. In particular there are two-letter mnemonics for the basic instruction types, and letters can be added to these to specify the type of addressing, to select the complement of the source in data transmission, and to specify particular operators for function generating and testing. The basic form for assembling any instruction word is

```
T     S,F,D
```

where *T* is the mnemonic for the instruction type, *S* and *D* are the source and destination, and *F* represents the function bits in the middle of the instruction word. For all one-word instructions (those that do not reference memory), *S* and *D* are 2-digit octal operator codes. An operator code indicating memory reference is always absorbed in *T*, and a memory address then replaces the operator code in the *S* or *D* position or both. Consider the instruction that moves the word in register AX to register AY shifted right one place. In mnemonic form this would be

```
RR    AX,R1,AY
```

which assembles as 11 1100 12. To move a word from memory location 317 to register AX would be

```
MR    317,11
```

or

```
MR    317,AX
```

either of which assembles as two consecutive words, 06 0000 11 and 000317.

Note

Numbers representing instruction words always have a pair of octal digits at each end for the source and destination operator codes and four binary digits between them for the function bits. All numbers representing codes, addresses, and register contents except instruction words are always octal, and any numbers appearing in program examples are octal unless otherwise specified. Computer words (other than instruc-

tions) are represented by six octal digits wherein the left one is always 0 or 1 representing the value of bit 15. The ordinary use of numbers in the text to count steps in an operation, to specify word or byte lengths, bit positions, exponents, etc, or to specify quantities of objects such as words or locations, employs standard decimal notation.

In all but the nonmemory data transmission instructions, one or both of the operator codes is implied by the mnemonic for the instruction type. Hence the basic mnemonic not only represents the numberical value but also tells the assembler how to interpret the rest of the information given for the instruction. Since the format thus varies from one instruction type to another, the detailed conventions for programming are given with the descriptions of the various types in Chapter 2.

The programming examples in this manual use the following addressing conventions:

A colon following a symbol indicates that it is a symbolic location name.

```
A:    RR    10,04
```

indicates that the location that contains RR 10,04 may be addressed symbolically as A.

The period represents the current address, *eg*

```
      RM    10,.+4
```

is equivalent to

```
A:    RM    10,A+4
```

Anything written at the right of a semicolon is commentary that explains the program but is not part of it.

1.4 SOFTWARE

Software for the GRI-909 includes the following:

FAST — a functional assembler for the system-oriented programming language described in Appendix A.

BASE — a basic assembler for the language described in the body of this manual.

BASE 360 — a version of BASE written in PL/1 for the IBM 360 series computers.

EDIT — a source tape editor combining line-oriented and content-oriented editing commands.

DEBUG — a debugging program.

Fixed point routines for single and double precision arithmetic.

An interpretive floating point package for basic floating point arithmetic. An extended package includes additional sophisticated mathematical functions.

Input-output routines for the standard peripherals.

A series of data conversion routines for converting from external to internal form and vice versa for single and double precision fixed point integers and fractions, BCD integers and fractions, and floating point numbers.

Diagnostics.

CHAPTER TWO

DIRECT FUNCTION PROCESSOR

This chapter describes the instruction types in detail, describes the use of the instructions for the basic operators including the program interrupt and console, and presents a general discussion of input-output. The effects of instructions that address particular peripheral devices and optional functional operators are discussed with the operators in the remaining chapters.

The description of each instruction begins with the mnemonic, the name, and a box showing the format. The mnemonic assembles to the word in the box, where bits in those parts of the word represented by letters assemble as 0s. The letters indicate portions that must be added to the mnemonic to produce a complete instruction word.

The descriptions of the basic instruction types in §§2.2-2.5 give the processor execution time. This is always a multiple of the processor cycle time of 1.76 μs. Each processor cycle includes one memory read-write cycle, and every instruction in the stored program requires a minimum of one cycle simply to retrieve the instruction from memory. Instructions supplied by an external source cannot reference memory and are executed at the rate of two per processor cycle. Since instructions described in the rest of the manual are simply particular instances of the basic types, no times are given except for function generating instructions that start operators that in turn preempt the processor, *ie* stop the stored program.

Number System. In any arithmetic operation the hardware treats computer words as 16-bit unsigned binary numbers in the range 0 to $2^{16} - 1$. But the programmer can interpret these as signed numbers using the equivalent twos complement conventions.

In a word used as a signed number, bit 15 (the leftmost bit) represents the sign 0 for positive, 1 for negative. In a positive number the remaining fifteen bits are the magnitude in ordinary binary notation. The negative of a number is obtained by taking its twos complement. If x is an n-digit binary number, its twos complement is $2^n - x$, and its ones complement is $(2^n - 1) - x$, or equivalently $(2^n - x) - 1$. Subtracting a number from $2^n - 1$ (*ie* from all 1s) is equivalent to forming the logical complement, *ie* changing all 0s to 1s and all 1s to 0s. Therefore, to form the twos complement one takes the logical complement — usually referred to merely as the complement — of the entire word including the sign, and adds 1 to the result. In a negative number the sign bit is 1 and the remaining bits are the twos complement of the magnitude.

Operations on signed numbers using twos complement conventions are identical to operations on unsigned numbers; the hardware simply treats the sign as a more significant magnitude bit. Suppose we wish to count seventeen steps by incrementing during data transmission. We would start with a register containing $2^{16} - 17$, *ie* this binary configuration,

$+141_{10} = +215_8 =$ 0 000 000 010 001 101 (15 … 0)

$-141_{10} = -215_8 =$ 1 111 111 101 110 011 (15 … 0)

1 111 111 111 101 111 (15 … 0)

and increment until overflow occurs at 2^{16}. As an unsigned number the above would be equivalent to

$$177757_8 = 32751_{10}$$

whereas interpreted as a signed number using twos complement notation it would be

$$-21_8 = -17_{10}$$

(2^{16} - 17 is the twos complement of 17). Hence we can regard the count as starting at a large number with overflow at 2^{16}, or as starting at a small negative number with overflow at zero. Insofar as processor operations are concerned, it makes no difference which way the programmer interprets the contents of various registers provided only that he is consistent. For further information on the properties of twos complement numbers and on the number formats for the software, refer to Appendix H.

Since each bit position represents a binary order of magnitude, shifting a number is equivalent to multiplication by a power of 2, provided of course that the binary point is assumed stationary. Shifting one place to the left multiplies the number by 2. A 0 should be entered at the right, and no information is lost if a 0 is shifted out at the left (for a signed number the condition is that the sign bit remain the same — a change in the sign indicates that a bit of significance has been shifted out). Shifting one place to the right divides by 2. Truncation occurs at the right, and a 0 must be entered at the left (for a signed number a bit equal to the sign must be entered).

2.1 OPERATORS

The instruction format allows sixty-four operator codes. The table opposite lists the octal codes and mnemonics for the basic operators that are part of every system, the arithmetic operator, and some of the standard peripheral devices. A complete list of operator codes is included in Appendix E.

Sometimes a single code may name two operators, one as a source, another as a destination. Thus the code 02 as a source specifies the function generator, but as a destination it specifies the function tester. In some cases a code cannot be used with all instruction types, and every code generally has different meanings with different instruction types. *Eg* the code 00 specifies the control logic when used with function generation or function testing, but it is a null code when used with data transmission (*ie* as a source it supplies a zero word, and as a destination it can receive no data). Similarly the code 04 represents interrupt control elements when used with function generation or testing, but represents the interrupt status register when used for a data transfer.

Consider the arithmetic operator, which has three codes, two for the registers in it and one for the operator itself. The registers can be addressed as source or destination of data, but no functions can be generated for them nor do they supply any functions for testing. The operator on the other hand can receive functions — specifically the arithmetic or logical operation to be performed — and it has a flag that can be tested. It can also be a source of data for it supplies the result of the arithmetic operation, but it can receive no data as destination. Similarly all in-

OPERATOR CODES

Octal	Mnemonic	Operator	
		Source	Destination
00		Null-Control	
01		Instruction Register	Null
02		Function Generator	Function Tester
03	TRP	Trap Register	Data Tester (*nonmemory source*) Trap Register (*memory source*)
04	ISR	Interrupt Status Register	
05		Memory Address	Null
06		Memory (Buffer)	
07	SC	Sequence Counter	
10	SWR	Console Switch Register	Null
11	AX	Register AX	
12	AY	Register AY	
13	AO	Arithmetic Operator	
14	MPO	Multiply Operator	
15		External Data	
16		External Address	Null
17	MSR	Machine Status Register	
75	RTC	Real Time Clock	
76	HSR HSP	Paper Tape Reader	Paper Tape Punch
77	TTI TTO	Teletype Input	Teletype Output

out devices use both types of function instructions, but an output device generally cannot supply data as a source, and an input device generally cannot receive data as a destination. Addressing as a source an operator that cannot supply data, or addressing as a destination an operator that cannot receive it, is equivalent to using the null code. However, pairs of IO devices sometimes share a common code; examples are teletype input and output, and paper tape reader and punch. In each case the code addresses an input buffer as source in data transmission or data testing, an output buffer as destination in data transmission, and the control logic for both devices as source or destination in function generation or testing.

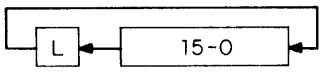
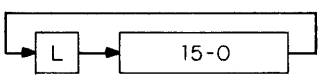
Some codes are used only for the internal operations of the processor and cannot be used by the program to address the operators they designate. An attempt by the program to send data to the instruction register, the memory address register, or the external data or address registers results simply in a no-op. (The codes 15 and 16 are used for direct memory access for all devices and select individual registers on a priority basis. The program can actually address the external address register in any given device, but a

unique code (not 16) is assigned to each register.) The program can read the current memory address and the instruction (which is simply itself), but using code 15 or 16 as a source gives a zero word.

2.2 DATA TRANSMISSION

Any instruction moves a word of data from one place to another provided only that meaningful operator codes are used: no transfer occurs if the programmer specifies an operator that results in an instruction of some other type (such as a function instruction) or uses a code that is not available to the program.

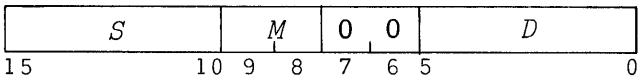
The word being moved can be modified in transit: the program can increment it by one or can shift it one place to the left or right. Associated with these functions are a Bus Overflow flag, and a 1-bit link register which connects the ends of the word for a circular shift — in other words the shift is actually a rotation through the link. The modification is selected by bits 8 and 9 of the instruction as follows.

Bits 9-8	*Mnemonic*	*Modification*
0		None
1	P1	Add +1. If the result is 2^{16} set Bus Overflow, otherwise clear it.
2	L1	Rotate left one place. Bit 15 is shifted into the link, the link into bit 0. 
3	R1	Rotate right one place. Bit 0 is shifted into the link, the link into bit 15. 

There are two basic instructions for transmission from one nonmemory register to another [*transmission instructions that reference memory are discussed as a special case below*]. One instruction takes a word from the source directly, the other takes its complement. System language statements for these instructions are of the form

Register X (Modification) TO *Register Y*

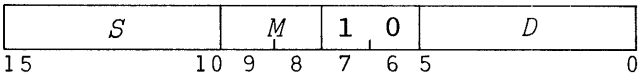
RR Register to Register
1 cycle 1.76 µs



Take the word from register *S*, modify it as specified by *M* (as given above), and place the result in register *D*. The contents of *S* are unaffected, the original contents of *D* are lost.

Add 1 to SC so the next instruction will be taken from the next location.

RRC Register to Register, Complement
1 cycle 1.76 µs



Take the complement of the word from register *S*, modify it as specified by *M* (as given above), and place the result in register *D*. The contents of *S* are unaffected, the original contents of *D* are lost.

Add 1 to SC so the next instruction will be taken from the next location.

Suppose we wish to load register AX with the address of the location containing the instruction being executed. We could give

```
RR    7,11
```

which addresses SC and AX as source and destination. To load twice the current

address we would give

```
RR    7,2,11
```

or, more likely, the mnemonic form

```
RR    SC,L1,AX
```

which assembles as 07 1000 11. To load the twos complement of AX into AY we would give

```
RRC   AX,P1,AY
```

which assembles as 11 0110 12.

For convenience the assembler recognizes special mnemonics for clearing a register (loading zero into it) and transferring a register into itself.

ZR *M*,*D* is equivalent to RR 0,*M*,*D*

RS *S*,*M* is equivalent to RR *S*,*M*,*S*

In both cases the letter C can be appended to the basic mnemonic to take the complement of the source. To load +1 into AX we would give

```
ZR    P1,AX
```

and to load all 1s we would give

```
ZRC   AX
```

To change the word in AX to its twos complement this suffices:

```
RSC   AX,P1
```

Memory Reference

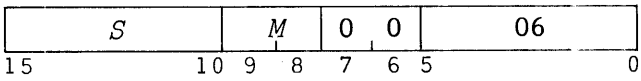
A special case of data transmission is instructions that specify memory as the source or destination of data. The programmer need specify only the non-memory operator, but must supply an operand itself for the location following the instruction. (Of course the second location can be left clear if an immediate operand is to be placed in it.) There are eight basic instruction forms for the two transfer directions, each with four modes of addressing. System language statements for them are like this.

Register (Modification) TO *Location*

Location (Modification) TO *Register*

RM Register to Memory

3 cycles 5.28 µs

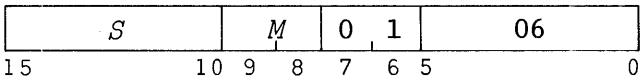


Add 1 to SC to retrieve the effective address *E* from the next location. Add 1 to SC again so the next instruction will be taken from the second location following this instruction.

Take the word from register *S*, modify it as specified by *M* (as given above), and place the result in location *E*. The contents of *S* are unaffected, the original contents of location *E* are lost.

RMD Register to Memory, Deferred

4 cycles 7.04 µs

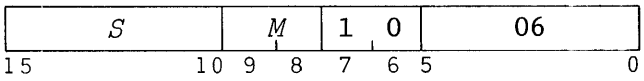


Add 1 to SC to retrieve the indirect address *I* from the next location. Add 1 to SC again so the next instruction will be taken from the second location following this instruction. Retrieve the word from location *I* and add 1 to it to produce the effective address *E*. Store *E* in location *I*.

Take the word from register *S*, modify it as specified by *M* (as given above), and place the result in location *E*. The contents of *S* are unaffected, the original contents of location *E* are lost.

RMI Register to Memory, Immediate

2 cycles 3.52 µs

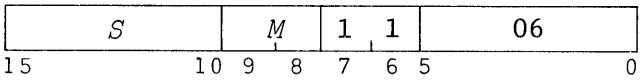


Add 1 to SC to produce the effective address *E*. Take the word from register *S*, modify it as specified by *M* (as given above), and place the result in location *E* (*ie* the next location). The contents of *S* are unaffected, the original contents of location *E* are lost.

Add 1 to SC again so the next instruction will be taken from the second location following this instruction.

RMID — Register to Memory, Immediate and Deferred

3 cycles — 5.28 µs

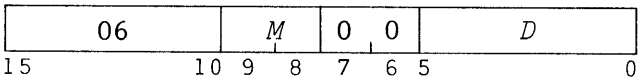


Add 1 to SC to produce the indirect address *I*. Retrieve the word from location *I* (*ie* the next location) and add 1 to it to produce the effective address *E*. Store *E* in location *I*. Add 1 to SC again so the next instruction will be taken from the second location following this instruction.

Take the word from register *S*, modify it as specified by *M* (as given above), and place the result in location *E*. The contents of *S* are unaffected, the original contents of location *E* are lost.

MR — Memory to Register

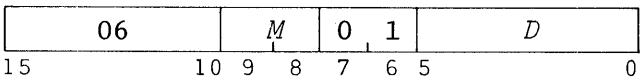
3 cycles — 5.28 µs



Perform the same address operations as RM. Then take the word from location *E*, modify it as specified by *M* (as given above), and place the result in register *D*. Location *E* is unaffected, the original contents of *D* are lost.

MRD — Memory to Register, Deferred

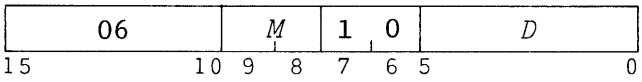
4 cycles — 7.04 µs



Perform the same address operations as RMD. Then take the word from location *E*, modify it as specified by *M* (as given above), and place the result in register *D*. Location *E* is unaffected, the original contents of *D* are lost.

MRI — Memory to Register, Immediate

2 cycles — 3.52 µs

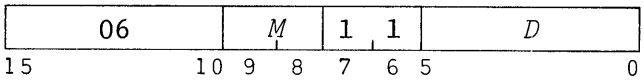


Add 1 to SC to produce the effective address *E*. Take the word from location *E* (*ie* the next location), modify it as specified by *M* (as given above), and place the result in register *D*. Location *E* is unaffected, the original contents of *D* are lost.

Add 1 to SC again so the next instruction will be taken from the second location following this instruction.

MRID — Memory to Register, Immediate and Deferred

3 cycles — 5.28 µs



Perform the same address operations as RMID. Then take the word from location *E*, modify it as specified by *M* (as given above), and place the result in register *D*. Location *E* is unaffected, the original contents of *D* are lost.

The assembly format for memory reference is the same as for other data transmission instructions except that the contents of the location following the instruction replace the unneeded code for the memory operator.

```
MR    542,AX
```

which assembles into the two consecutive words 06 0000 11 and 000542, places the contents of location 542 in AX.

```
RMI   AX,R1,0
```

assembles as 11 1110 06 and 000000, and stores the contents of AX divided by 2 into the location following the instruction (the location left clear in the assembly).

For convenience the assembler recognizes special mnemonics for clearing a location and transferring a location into itself. Letting *W* be the word given for the second location,

ZM *M,W* is equivalent to RM 0,*M,W*

MS *W,M* is equivalent to MR *W,M*,6

In both cases the letters D, I and ID can be appended to the basic mnemonic to select deferred, immediate, and immediate-deferred addressing. To simply keep a count of thirty in the location following an instruction we could give

```
MSI   -36,P1      ;30₁₀ = 36₈
```

which assembles as 06 0110 06 and 177742. The thirtieth iteration of this instruction sets the Bus Overflow flag.

Addressing Examples. Suppose AX contains 000132 and the following locations contain the numbers listed.

Location	*Contents*
320	001742
1742	005360
1743	134267
5361	000023

Then executing these instructions in location 317 produces the effects given.

Instruction		*Effect*
MR	1742,AX	Load 5360 in AX.
MRD	1742,AX	Change location 1742 to 5361 and load 23 in AX.
MRI	1742,AX	Load 1742 in AX.
MRID	1742,AX	Change location 320 to 1743 and load 134267 in AX.
RM	AX,1742	Store 132 in location 1742.
RMD	AX,1742	Change location 1742 to 5361 and store 132 in location 5361.
RMI	AX,1742	Store 132 in location 320.
RMID	AX,1742	Change location 320 to 1743 and store in location 1743.
MS	1742,R1	Change location 1742 to 2570 (5360÷2).
MSD	1742,R1	Change location 1742 to 5361 and change location 5361 to 11 (23÷2).
MSI	1742,R1	Change location 320 to 761 (1742÷2).
MSID	1742,R1	Change location 320 to 1743 and change location 1743 to 56133 (134267÷2).
ZM	P1,1742	Store 1 in location 1742.
ZMD	P1,1742	Change location 1742 to 5361 and store 1 in location 5361.
ZMI	P1,1742	Store 1 in location 320.
ZMID	P1,1742	Change location 320 to 1743 and store 1 in location 1743.

2.3 DATA TESTING

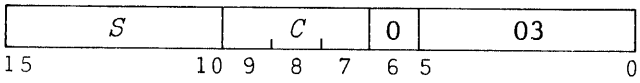
Using codes 03 and 06 as source and destination or vice versa produces a standard data transfer between memory and the trap register. Thus such instructions as MR 100,TRP and RMI TRP,0 can use TRP simply as a general purpose register. But code 03 as destination with a *nonmemory* source addresses the data tester, allowing the program to perform an arithmetic test on the source word interpreted as a signed number. If the number satisfies the condition specified by the instruction, a jump is executed and a return address is saved in the trap register.

There are two forms of the data testing instruction using direct and deferred addressing. In the system language they have the form

IF *Operator Condition* GO TO *Location*

JC — Jump Conditional

If no jump: 1 cycle — 1.76 μs
If jump: 2 cycles — 3.52 μs



If *S* is *not* 06, test the word from register *S* for the condition specified by *C* (the contents of *S* are not affected by the test).

Bit	*Effect of a 1 in the bit*
9	Selects the condition that the source word is less than zero (*ie* bit 15 of the word is 1).
8	Selects the condition that the source word is zero (*ie* its bits are all 0s).
7	Inverts the conditions selected by bits 8 and 9. A 0 in bit 7 selects the OR of the conditions selected by 8 and 9; a 1 in bit 7 selects the AND of the complements of those conditions.

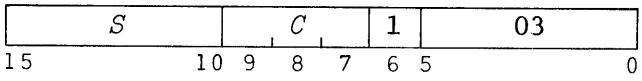
The combined effects of bits 7-9 and the mnemonics for the various bit configurations are as follows.

Bits 9-7	*Mnemonic*	*Jump function*
0		Never Jump
1		Always Jump
2	ETZ	Jump if Equal to Zero
3	NEZ	Jump if Not Equal to Zero
4	LTZ	Jump if Less than Zero
5	GEZ	Jump if Greater than or Equal to Zero
6	LEZ	Jump if Less than or Equal to Zero
7	GTZ	Jump if Greater than Zero

If the word from *S* does not satisfy the specified condition, add 2 to SC so the next instruction is taken from the second location following this instruction. If the condition is satisfied, add 1 to SC, load the incremented SC into the trap register and also use it to retrieve the effective address *E* from the next location. Load *E* into SC to retrieve the next instruction from location *E* and continue sequential operation from there.

JCD — Jump Conditional, Deferred

If no jump: 1 cycle — 1.76 μs
If jump: 3 cycles — 5.28 μs



If *S* is *not* 06, test the word from register *S* for the condition specified by *C*, as described above under JC. If the word from *S* does not satisfy the condition, add 2 to SC so the next instruction is taken from the second location following this instruction.

If the condition is satisfied, add 1 to SC, load the incremented SC into the trap register and also use it to retrieve the indirect address *I* from the next location. Retrieve the word from location *I*, add 1 to it to produce the effective address *E*, and store *E* back in location *I*. Load *E* into SC to retrieve the next instruction from location *E* and continue sequential operation from there.

The assembly format for data testing is the same as for memory reference data transmission except that the address in the location following the instruction replaces the destination code for the data tester, which is implied by the instruction mnemonic. To jump to location NEG for a subroutine that handles a negative result of an arithmetic operation, this suffices:

```
JC    AO,LTZ,NEG
```

For convenience the assembler recognizes a special mnemonic for an unconditional jump, *ie* one that specifies a condition that is necessarily satisfied. Letting A be the address given for the second

Example 1.1

```
        JU    SUB             ;2 words for subroutine call
        .                     ;N words of data
        .
        .
        0                     ;A zero word terminating the data
        ...                   ;Next instruction
```

Example 1.2

```
SUB:    RMI   TRP,0           ;Save original SC in next location
NEXT:   MRD   SUB+1,TRP       ;Get next data word
        JC    TRP,ETZ,END     ;Is word zero?
        .                     ;No, process data
        .
        .
        JU    NEXT            ;Get next word
END:    .                     ;Yes, complete data processing
        .
        .
        JUD   SUB+1           ;Return to calling sequence
```

location,

JU A is equivalent to JC 0,ETZ,A

JUD A is equivalent to JCD 0,ETZ,A

Note that when SC is saved it points to the address location following the jump instruction. A subsequent return to the next instruction in the calling sequence (to the second location following the jump) can therefore be made by giving

```
RR    TRP,SC
```

as SC is automatically incremented following the transfer. Note also that SC is saved in the trap register. Hence the subroutine can be reentrant (pure), *ie* memory is not modified by the act of calling it. If we wish to have the trap free during the subroutine, then at the subroutine entry point we can use

```
ENT:  RMI   TRP,0
```

which moves the address from the trap to the location following ENT. The return can then be made by giving

```
JUD   ENT+1
```

The above technique is also convenient for argument passing. Suppose we wish to call a routine to process *N* words of (nonzero) data and give the words with the call. The calling sequence can be like Example 1.1, and the subroutine would then be of the form of Example 1.2.

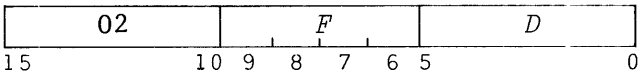
2.4 FUNCTION GENERATION

By addressing the function generator as source (code 02) the program can perform functions in the destination operator. Physically this is done by pulsing control lines selected by 1s in bits 6-9 of the instruction, which has this form in the system language.

Function TO *Operator*

FO — Function Output

1 cycle — 1.76 μs



Perform the functions specified by *F* in operator *D*. In some cases individual functions are selected by 1s in specific bits in *F*; in other cases the various configurations of a set of bits in *F* select

specific functions (*eg* the four configurations of bits 8 and 9 select the four arithmetic and logical functions that can be performed by the arithmetic operator). In any event the actual functions that can be selected and the manner in which they are selected by bits 6-9 depend on the properties of operator *D*.

Since the source code is implied by the mnemonic FO, assembly statements for the function generating instructions are of the form

FO *F,D*

The assembler recognizes special mnemonics that include the codes for the more common destination operators: the machine (*ie* the control logic), and the interrupt and arithmetic operators.

FOM *F*	is equivalent to	FO *F*,0
FOI *F*	is equivalent to	FO *F*,ISR
FOA *F*	is equivalent to	FO *F*,AO

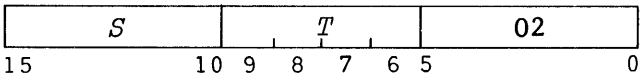
There are also mnemonics for programming bits 6-9, and functions can be combined (when it is meaningful to do so) simply by giving the appropriate mnemonics separated by spaces.

2.5 FUNCTION TESTING

By addressing the function tester as destination (code 02) the program can test control signals from the source operator for a skip. The system language statement for this instruction is

SKIP IF *Operator Function*

SF — Sense Function
1 cycle — 1.76 µs



Perform the test *T* on functions from operator *S*. Individual functions (control signals, flags) from *S* are selected by 1s in bits 7-9. If bit 6 is 0, the test is positive if *any* function selected by bits 7-9 is true. If bit 6 is 1, the test is positive if *no* function selected by bits 7-9 is true.

If the test is negative, add 1 to SC so the next instruction is taken from the next location. If the test is positive, add 3 to SC to skip the next two locations in normal sequence. In other words the processor takes the next instruction from the third location following this instruction — two locations are skipped over so that the skipped instruction can be a jump.

Since the destination code is implied by the mnemonic SF, assembly statements for the function testing instructions are of the form

SF *S,T*

The assembler recognizes special mnemonics that include the codes for the more common source operators: the machine (control logic) and the arithmetic operator.

SFM *T*	is equivalent to	SF 0,*T*
SFA *T*	is equivalent to	SF AO,*T*

There are also mnemonics for programming the functions in bits 7-9, and functions can be combined simply by giving the appropriate mnemonics separated by spaces. The mnemonic NOT preceding the function mnemonics inverts the test, *ie* it places a 1 in bit 6. Hence

SFM BOV LNK

which assembles as 00 0110 02 (SFM 6), skips the next two locations if either Bus Overflow or the link is set. But

SFM NOT BOV LNK

which assembles as 00 0111 02 (SFM 7), skips if neither Bus Overflow nor the link is set.

Function testing instructions for all operators act in the manner indicated above, so the descriptions of these in-

structions in the remainder of the manual simply list the functions tested. In each format box, bit 6 is represented by the letter *N*.

Note that a function testing instruction skips the next *two* locations. Hence if the instruction to be skipped uses only one location, the programmer must fill in with a no-op. The standard no-op is simply 00 0000 00, for which the assembler recognizes the mnemonic NOP. Suppose we wish to twos complement AX if the link is set, but otherwise simply complement it before storing it in memory. We would complement in either case, but test to skip the incrementing by one like this:

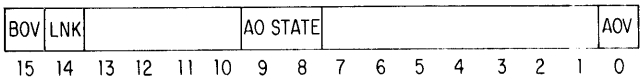
```
RSC   AX
SFM   NOT LNK
RS    AX,P1
NOP
RM    AX,M
```

2.6 PROGRAM CONTROL

The present section discusses the use of the various instruction types to control the program sequence, the transmission operator, the basic state of the computer, and the execution of external instructions. The use of the jump instructions for handling subroutines is treated in §2.3. A jump always causes the next instruction to be taken from the address loaded into SC, but this is not always true when a data transmission instruction loads SC. The data transfer always occurs in the final cycle of the instruction, and SC is incremented in the first and second cycles. Hence if the instruction takes only one or two cycles (*ie* an RR or MRI), SC is incremented after it is loaded. In general this simplifies the return because the address saved in TRP must be incremented in order to point to the correct return location. This incrementing can also be handled by using a deferred jump, but the program must increment in the bus modifier when using a data transmission instruction of three or four cycles to return with an address originally saved in TRP.

Sending a word to an operator that cannot receive information or that does not exist is a no-op, which is effectively a program delay. The basic no-op NOP is a one-cycle null transfer. The length of the delay is equal to the number of cycles the instruction takes. Provided no modification is called for, many transmission instructions have no effect on the computer at all except for the regular SC incrementing to go to the next instruction; but the programmer must remember that deferred addressing does affect some memory location.

Certain flags and control flip-flops in the computer are connected to the source and destination buses in such a way that their states can be saved and then restored as though they constituted a register. These elements are referred to collectively as the machine status register, which can be addressed as operator code 17, mnemonic MSR. The elements that make up this register are the following.



Saving and restoring the machine state is a procedure used primarily in program interrupts [§2.8], but the programmer can use it anytime; he can even set up the machine state in any way he likes by loading a word of his own construction into MSR. Having the Bus Overflow and Arithmetic Overflow flags at the ends of the register is especially convenient for either of them can therefore be moved to the link in only one cycle. This is done by

```
RR    MSR,L1,0
```

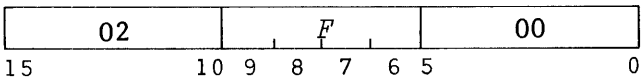
and

```
RR    MSR,R1,0
```

respectively.

There are also function generating and testing instructions for the processor control logic.

FOM Function Output, Machine

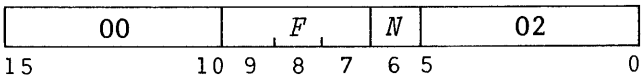


Perform the functions specified by 1s in *F* as follows.

Bit	*Mnemonic*	*Function*
6	CLL	Clear link
7	STL	Set link
8	HLT	Halt the processor

Programming 1s in bits 6 and 7, mnemonic CML, complements the link.

SFM Sense Function, Machine



Perform a function test (as described in §2.5) on the conditions selected by 1s in *F* as follows.

Bit	*Mnemonic*	*Condition*
7	BOV	Bus Overflow set
8	LNK	Link set
9	POK	Power ok

To determine whether location A contains all 1s we can use this instruction pair.

```
      MR    A,P1,0     ;Add 1, throw away
                       ;result
      SFM   BOV        ;Skip if overflow
                       ;occurred
```

Suppose we wish to use bit 0 of location A as a program flag. We can set it with this instruction:

```
      ZM    P1,A
```

and we can test it by giving this sequence:

```
      MR    A,R1,0     ;Put bit 0 in link
      SFM   LNK        ;Skip if link set
```

If the computer contains an accumulator or any data register that can be transmitted to itself, the twos complement of a word can be formed by a single RSC. Without such a register, the twos complement (say of the word in location Z) must be constructed one bit at a time, like this.

```
      MRI   -21,TRP    ;Set count to -17
      RM    TRP,M2+1
M1:   MS    Z,R1       ;Bit by bit rotate
      FOM   CML        ;Complement bit
M2:   MSI   0,P1       ;Count step
      SFM   BOV        ;Finished ones
                       ;complement?
      JU    M1         ;No, do next bit
      MS    Z,P1       ;Yes, add 1
```

External Instructions

Operations in some functional operators are limited to single data transfers or state changes, and they thus take place entirely within the instructions that cause them. But in many cases an FO instruction for an operator can trigger an operational sequence that delays execution of the stored program until it is finished. There are two types of operators that do this: one stops the processor while it executes its own internal operations, the other takes control of the processor to execute a sequence of external instructions sent to IR and retrieved from its own built-in read-only memory. In either case the sequence is always started by an FO instruction that addresses the operator as destination and has a 1 in bit 6 (programmed by the mnemonic STRT). A sequence of external instructions takes control of the processor in order to use other operators, such as the arithmetic operator, or at least the data transfer paths. In some respects such a sequence can be regarded as an extension of the FO that triggered it, for SC remains constant, and program interrupts are shut out until the sequence is finished. But following any cycle the processor can pause for direct memory access. Thus high speed in-out operations are not endangered except to the extent that a program interrupt may be delayed.

2.7 INPUT-OUTPUT

With direct function processing, an in-out instruction is simply one that addresses an in-out operator, *ie* a peripheral device. A table in Appendix F lists all devices for which operator codes have been assigned, and gives their mnemonics and GRI option numbers.

Besides the standard selection nets for decoding source and destination addresses, every device operator has a Ready flag and an Interrupt Status flag. The first of these denotes the state of the device. At power turnon, all output Ready flags are set, all input Ready flags and Interrupt Status flags are clear. Placing a device in operation clears Ready. If the device will be used for input, the program places it in operation by giving an FO instruction. A complex device to be used for output may require an FO, but a simple output device is usually started automatically by giving a data transmission instruction that sends a unit of data — a word or character depending on how the device handles information. (The word "output" used without qualification always refers to the transfer of data from the bus system to the peripheral equipment; "input" refers to the transfer in the opposite direction.) When the device has processed a unit of data, it sets Ready to indicate that it is ready to receive new data for output, or that it has data ready for input. In the former case the program would respond with a transmission instruction to send more data; in the latter with a transmission instruction to bring in the data that is ready, followed by an FO to restart the device. If the program has set the Interrupt Status flag, the setting of Ready signals the program by requesting an interrupt; if Interrupt Status is clear, then the program must keep testing Ready to determine when the device is available.

A consistent format is employed for FO and SF instructions for all devices. Except for simple output devices that start automatically when data is sent to them, a device is usually started by programming a 1 in bit 6 for an FO that addresses the device as destination. All input devices require this, and the mnemonic is STRT. (Note that this is the same bit that starts a functional operator, and the mnemonic is also the same.) Clearing or sensing of the Ready flag is generally done by a 1 in bit 7 of an FO or SF respectively. If a pair of devices — one output only, the other input only — share a common operator code, bit 9 handles the flag for input, bit 7 the flag for output. In other cases bits 8 and 9 may be used for special flags. The mnemonic IRDY places a 1 in bit 9, ORDY places a 1 in bit 7. It is usually convenient to clear the input Ready flag when starting the device, so the assembler recognizes INP as equivalent to IRDY STRT.

A device may require no data transfers, such as one type of real time clock that uses only an FO to turn it on and off. All of the simpler data handling devices have only one buffer, *eg* to hold a single character in the teletype, tape reader and tape punch, or to receive incremental plotting data for a single point in the plotter. A high speed device, such as magnetic tape or disk, may use data transmission instructions *only* for control information with data moving between the device and memory via direct memory access. Control information the program must supply to a tape system includes a transport address and an actual command the tape operator is to perform; input information includes error flags and transport status levels.

Most peripheral devices involve motion of some sort, usually mechanical. In this respect there are two types of devices, those that stay in motion and those that do not. Magnetic tape is an example of the former type. Here the device executes a command (such as read, write, space forward) and a ready flag indicates when the entire operation is finished. A separate data flag signals each time the device is ready for direct memory access, but the tape keeps moving until an entire record or file has been processed. Paper tape, on the other hand, stops after each line is read, but if the program restarts it within a critical time the tape moves

continuously.

Other devices operate in one or the other of these two ways but differ in various respects. The tape punch and teletype printer are like the reader. Teletype keyboard input is initiated by the operator striking a key rather than by the program. Once started the card reader reads an entire card, with a data transfer required for each column.

2.8 PROGRAM INTERRUPT

Most in-out devices must be serviced infrequently relative to the processor speed and only a small amount of processor time is required to service them, but they must be serviced within a short time after they request it. Failure to service within the specified time (which varies among devices) can often result in loss of information and certainly results in operating the device below its maximum speed. The program interrupt is designed with these considerations in mind, *ie* the use of interruptions in the current program sequence facilitates concurrent operation of the main program and a number of peripheral devices. The hardware also allows a power failure to signal the program by requesting an interrupt.

Interrupt Requests. Interrupt requests by a device are governed by its Ready and Interrupt Disable flags. When a device completes an operation it sets Ready, and this action requests a program interrupt if Interrupt Status has been set by the program — if Interrupt Status is clear the device cannot request an interrupt. At the beginning of every cycle the processor synchronizes any requests that are then being made. Once a request has been synchronized the device that made it must wait for an interrupt to start. The request signal is a level so once synchronized it remains on the bus until the program clears Ready or Interrupt Status. In other words clearing either flag in a device disables any request the device has alreay made and had synchronized, so it is no longer waiting for an interrupt. If the program clears Interrupt Status but leaves Ready set, subsequently setting the former flag again restores the request (remember the program cannot set Ready; only the device can do that).

Starting an Interrupt. The processor starts an interrupt if all four of the following conditions hold.

The processor has just completed an instruction or a direct memory access [§2.9]. Insofar as interrupts are concerned an entire sequence of external instructions is equivalent to a single instruction in the program: once a sequence has started, the processor does not handle any interrupts until it is finished.

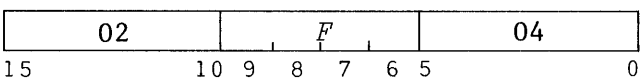
At least one device is waiting for an interrupt to start (*ie* it was requesting an interrupt at the beginning of the last cycle).

The interrupt control is on.

No device is waiting for direct memory access, *ie* there are no requests for such access that the processor has synchronized but not yet fulfilled. The direct memory channel has priority over program interrupts.

When the processor finishes an instruction it takes care of all direct memory requests before it starts an interrupt; this includes any additional direct memory requests that are synchronized while access is occurring. When no more devices are waiting for access, the processor starts an interrupt if the interrupt control is on and a device was requesting an interrupt at the beginning of the last access. The program governs the interrupt operator through this instruction.

FOI Function Output, Interrupt



Perform the functions specified by 1s in *F* as follows.

Bit	*Mnemonic*	*Function*
6	ICF	Turn interrupt control off
7	ICO	Turn interrupt control on

To start an interrupt the processor turns off the interrupt control so no further interrupts can be started, saves SC (which points to the next instruction) in a location whose address is supplied by the device, and loads SC with an address one greater than that supplied. The processor then goes on to the instruction in the location now addressed by SC and continues sequential operation from there. In general three locations are allocated to each channel to be used as follows.

The first location, whose address is supplied by the device, receives the current contents of SC.

The second location should contain the instruction 06 0010 07, *ie* an MRI -,SC.

The third location should contain an address one less than the first location in the service routine for the device.

The channel locations allocated to the basic in-out equipment are these.

Location	*Device*
11-13	Teletype output
14-16	Teletype input
17-21	High speed punch
22-24	High speed reader

Locations 25-62 are distributed into ten more channels for other devices [*Appendix F lists the interrupt locations for all GRI-supplied devices*]. A breakpoint [*see below*] or a power failure [§2.10] causes an interrupt to location 0, and the first six locations should be set up for these two combined channels this way.

Location	*Use*
0	SC stored here
1	Skip if power ok
2	MRI SC
3	Power failure routine start address − 1
4	MRI SC
5	Breakpoint routine start address − 1

In a large system it may be necessary to have two or more devices sharing a single channel; in such a case the third location must contain an address for a common routine for all of them. The hard-wired address in any device can be disabled so that it interrupts to location 0. If some but not all do so, then the instruction in location 4 should take the processor to a service routine for those devices (plus breakpoint). If all devices interrupt to 0, the service routine can begin right in location 1.

A device may actually interrupt directly to its service routine. In this case SC is stored in the first location of the routine, and the second location contains the first instruction of the routine.

Servicing an Interrupt. If more than one device is connected to a single channel, the service routine should first determine which one requires service; this is easily done by a series of SF instructions. Once the device has been identified, the routine should save the contents of any registers or flags that will be used in the routine or may be affected by it. Hence the routine should save the machine status register if there will be any modification in the bus modifier, as Bus Overflow or the link can be affected by such operations. Similarly TRP should be saved if the routine contains a jump or uses it as a general purpose register. Then the program should service the device. While doing so it can simply leave the interrupt off, or it can turn the interrupt back on and establish a priority structure that allows higher priority devices to interrupt the current device service routine. This priority is determined by controlling the states of the Interrupt Status flags in the various devices.

If this final course is taken and the program enables another device on the same channel, the routine must save the SC location so the return address to the interrupted program will not be lost should another interrupt occur on that channel.

Device Priority. There are several ways in which priorities are determined for or assigned to devices. An elementary priority is established by the hardware for devices that are requesting interrupts simultaneously in that the processor brings in a channel address from one and only one device: among those that are waiting it takes the address from that one which is physically closest to the processor on the bus. This however applies only to those devices that are waiting at the time an interrupt is started. Using SFs to determine which device to service establishes a priority by the order in which the devices are tested, but again this applies only to those that are waiting at the time.

The most significant method is by controlling the Interrupt Status flags to specify which devices can interrupt a service routine currently in progress. These flags are each connected to a particular data line in the source and destination buses, so collectively they constitute the interrupt status register. By addressing this register as operator code 04, mnemonic ISR, the program can save the current priority structure, establish a new one, or restore a previous one. In general the devices are in order by speed, with the fastest ones (those requiring the quickest service) assigned to the higher numbered bits in ISR, but there is no established priority as the program can set up any ISR configuration. All devices whose Interrupt Status flags are clear cannot cause an interrupt to start (clearing the flag causes the withdrawal of any request that has already been made and prevents the setting of Ready from making a request) and are therefore regarded by the program as being of lower priority. Those devices in which Interrupt Status is set can interrupt the current routine and therefore are regarded by the program as being of higher priority.

The following lists the devices assigned to the bits in ISR, and for each gives the mask that must be loaded into ISR to allow only devices assigned to higher numbered bits to interrupt. [*Complete information on all devices is given in Appendix F.*]

ISR Bit	*Device*	*Mask*
0	Teletype output	177776
1	Teletype input	177774
2	High speed punch	177770
3	High speed reader	177760
4		177740
5		177700
6		177600
7		177400
8		177000
9		176000
10		174000
11		170000
12		160000
13		140000
14		100000
15		000000

A ZRC ISR sets all the flags, allowing all devices to interrupt; a ZR ISR clears them all.

By means of ISR the program can establish any priority structure with one limitation: two or more devices whose flags are the same bit in ISR (*ie* are connected to the same data line) are all at the same priority level. When an interrupt is in progress for a device, the rest of the devices assigned to the same bit must be regarded as all of higher priority or all of lower priority depending upon whether they are enabled or not.

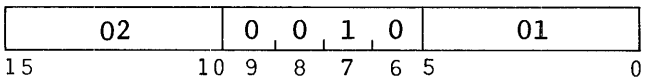
Dismissing an Interrupt. After servicing a device the routine should restore the pre-interrupt states of any operators affected by the routine (MSR, TRP, general registers), turn on the interrupt, and return to the interrupted program. The instruction that turns the interrupt back on has no effect until the next instruction begins. Thus after the FOI ICO the processor always executes one more instruction (assumed to be the return to the interrupted program) before another inter-

rupt can start. The return is usually an MR *L*,SC where *L* is the interrupt location for the channel, *ie* the location in which SC was saved when the interrupt was started.

If the service routine allows interrupts by higher priority devices, then before dismissing as indicated above, the routine should turn off the interrupt to prevent further interrupts during dismissal. In dismissing, the routine should reenable lower priority devices that were not allowed to interrupt the current routine but will be allowed to interrupt the program to which the processor is returning.

Breakpoint. The program can cause an interrupt at any time by the use of this instruction.

FO 2,1 Function Output, Breakpoint



Request an interrupt to location 0 and force the processor to accept it even if the interrupt control is off.

The interrupt requested by this instruction has priority over all others. After executing this instruction, the processor will first handle any direct memory requests that are waiting, but as soon as the last direct memory access is completed, an interrupt starts at location 0 even if the control is off. Starting the interrupt automatically removes the breakpoint request, but as previously indicated by the instructions recommended for locations 1-5, it should be assumed that a breakpoint has occurred if there is no power failure.

Obviously breakpoints are not used in any normal programming situation. They are in fact used almost exclusively for debugging purposes.

Timing. The time a device must wait for an interrupt to start depends on how many devices are using interrupts, how long the service routines are for devices of higher priority, and whether the direct memory channel is in use. A single device will shut out all others of lower priority if every time its service routine dismisses the interrupt, it is already waiting with another request; and the direct memory channel can preempt all processor time. If the channel is not in use the highest priority device need never wait longer than the time required for the processor to finish the instruction that is being performed when the request is synchronized. Maximum waiting time for ordinary instructions including synchronization is therefore about 9 μs. However the maximum possible time can be much longer if the program uses optional functional operators that stop the stored program. *Eg* multiplication can delay an interrupt by 56 μs when done by external instructions.

The time the processor takes to start an interrupt and jump to a service routine is three cycles, about 5 μs. An interrupt directly to a routine takes only the single cycle for storing SC.

When to Use the Interrupt. If the program has little computing to do and is using only one or two fast in-out devices or several slow ones, it may not be necessary to use the interrupt at all. On the other hand, if there are many calculations to perform and the program is using a fast device or is processing data using several slower devices, then the interrupt is necessary. The critical factors in determining whether to use the interrupt, and beyond that an ISR priority scheme, are what the program is doing besides in-out and the time required by the service routines. Suppose the program is doing nothing but processing data using reader, punch and teletype, and further suppose that no service routine requires more than say half a millisecond. In these circumstances the program could dispense with the interrupt and test all the devices (in the order just stated) by a loop containing SF and jump instructions, where

the reader service routine returns to the punch test and all others return to the beginning of the loop. The fastest device, the reader, will never be delayed too much. But suppose the program has a significant amount of computing to do. Then we must use the interrupt, but what about priority? If input-output service for the teletype requires .8 ms and punch service requires .5 ms, then the reader service will never be delayed too much if we simply turn the interrupt off while servicing each device. But if teletype service requires 20 ms per character, then neither reader nor punch will be able to run at full speed unless we use ISR to set up priority levels.

Programming Suggestions. A convenient method for handling a large number of priority levels is to use a pushdown list for saving the machine state. This obviates setting aside so many specific locations for saving MSR, TRP and various registers, and makes it very easy for a routine at any level in a sequence of nested routines to restore the state for the interrupted program.

Remember the following when programming an interrupt routine:

An interrupt cannot be started until the current instruction is finished. Therefore be cautious when using functional operators with lengthy external instruction sequences if devices that require very fast service can request an interrupt.

If several levels of interrupts are allowed, save the current ISR and reload it to shut out devices of lower priority.

Save MSR, TRP, etc if they will be used by the routine. *Eg* a routine could begin like this.

```
SUBR: RMI   ISR,0
      RMI   MSR,0
      RMI   TRP,0
      :
      :
```

The principal function of an interrupt routine is to respond to the situation that caused the interrupt. *Eg* computations that can be performed outside the routine should not be included within it.

Before returning to the interrupted program, restore the pre-interrupt states of ISR, MSR, TRP, etc. If they were saved as indicated above, they could be restored like this.

```
      MR    SUBR+1,ISR
      MR    SUBR+3,MSR
      MR    SUBR+5,TRP
      :
      :
```

2.9 DIRECT MEMORY ACCESS

The maximum rate for data transfers between external devices and core memory could be no greater than 80,000 words per second if the transfers were executed under program control. To allow rates up to 568,000 the processor contains a direct memory channel through which data can be transferred automatically using only one processor cycle per 16-bit word. At lower rates the channel also frees processor time to allow execution of a program concurrently with data transfers for a device.

Besides the straightforward transfer of a word between memory and a device in either direction, the channel also allows a device to increment by one a word already in memory. The direct memory channel is used by devices requiring very high data transfer rates, such as magnetic tape or disk, and by devices that utilize the memory increment feature, such as a pulse height analyzer.

The program cannot affect the channel directly because there are no instructions for it; instead the program sets up the device to use it. When the device requires data service, it requests direct memory access. At the beginning of every cycle the processor synchronizes any requests that are then being made. As soon as the processor completes an instruction in the

program or a cycle of two instructions in an external instruction sequence, it takes care of all requests that have been synchronized or are synchronized while it is handling transfers. If several devices are waiting for service simultaneously, the first to receive it is the one that is physically closest to the processor on the bus. After taking care of all direct memory requests, the processor returns to an external instruction sequence if one is in progress; otherwise it starts an interrupt if a device is waiting for one, or resumes the execution of instructions.

Timing. The time a device must wait for access depends on when its request is made within an instruction and how many devices of higher priority are also requesting access. Once the processor starts handling requests, a given device must wait until all devices closer than it on the bus have been serviced: the highest priority device can preempt all processor time if it requests access at the maximum rate. At less than the maximum rate the closest device need wait no longer than the time required for the processor to finish the instruction that is being performed when the request is synchronized. Maximum waiting time including synchronization is therefore about 9 μs.

2.10 POWER FAILURE DETECTOR AND AUTORESTART

When ac power is turned on, memory is unaltered, all output Ready flags are set, other flags and control flipflops are clear, TRP, AX, AY and other registers are indeterminate, and if the autorestart switch is off, SC is clear and the computer is stopped. If ac power should fail while the computer is running, there is a delay of at least 100 μs before the processor shuts down. In doing so, the processor always completes a cycle and sequences power off so the contents of memory are unaffected. The power failure detector warns the program when power is failing by requesting an interrupt to location 0. Although there is no interrupt status flag for the detector, so the program cannot turn it off independently, the interrupt control must be on for a power failure to produce an interrupt. Before making any other tests when an interrupt starts at location 0, the program should give an SFM POK to skip if power is alright [§§2.6, 2.8].

If power does fail, the program should save TRP, MSR, ISR, and all registers, and then halt. The action taken by the processor when an adequate power level is restored depends on the autorestart switch located on the front panel of the power supply behind the console. If the switch is off, power comes back on with the machine stopped. If the switch is on, then a few seconds after power comes back on the processor begins executing instructions in normal sequence at location 6. Locations 6 and 7 should therefore contain a jump to a suitable restart routine.

2.11 OPERATION

The console is illustrated on page 1-5. The lights at the left display control conditions, the rows of lights at the right display the processor registers. Below the latter is a register of toggle switches through which the operator can supply addresses and data to the destination bus (the down position of a switch represents a 1). The register can be used in conjunction with some of the operating keys, and its contents can be read by the program by addressing it as source code 10.

In the row at the bottom left are the operating keys. All but SS and STOP are momentary-contact. Each switch produces its indicated function when pressed down, except for WRITE which is normally down and must be lifted up to write in memory. At the upper left is a pair of octal thumbwheels for selecting operator codes to be used with the data switches and lights.

At the bottom of the console are two key-operated rotary switches. Turning

the right one clockwise disables the operating keys so no one can interfere with the operation of the processor (the operator can still use the data switches to supply information to the program). Turning the left rotary switch clockwise turns on power. When power comes on the register lights bear no relation to the actual contents of the registers until the operator initializes them by performing some operation. However, if the autorestart switch is on, the processor will actually go into normal operation beginning at location 6. It is thus recommended that the operator turn on the STOP switch before turning power on. Then the computer will execute one instruction and stop, and the instruction execution will also initialize the lights (the address in SC will depend on the instruction).

Indicators. When any indicator is lit the associated flipflop is in the 1 state or the associated function is true. A few indicators display useful information while the processor is running, but most change too frequently and are therefore discussed in terms of the information they display when the processor has stopped.

The top four rows of lights at the right are installed only in the programmer's console. The top row displays the instruction being executed or the last instruction completed. The lights are organized for ease in reading the contents of IR, the six lights on each end being the source and destination addresses, the middle four lights being the control bits. The next three rows of lights in order below IR display the contents of SC, MA and MB. The MA lights indicate the address to which the last memory access was made, the MB lights display the last data transmitted to MB. The data display lights at the bottom are used in conjunction with the operating keys, but while the program is running they display any information sent to the destination selected by the thumbwheels. This allows the operator to monitor any destination, or by selecting an unused code, it allows the program to supply information to the operator without affecting any internal register.

The lights at the left display the following control conditions.

FI	The next processor cycle will be used to fetch an instruction from memory.
FA	The next processor cycle will be used to fetch the address or process an immediate operand in a memory reference instruction.
FO	The next processor cycle will be used to process the operand in a memory reference data transmission instruction, or to fetch the second address in a deferred memory reference instruction of any type.
FD	The next processor cycle will be used to process the operand in a deferred memory reference data transmission instruction.
BK	The next processor cycle will be used to start an interrupt (break) by storing SC in the location addressed by the interrupting device.
DM	The next processor cycle will be used for direct memory access.
EI	The next processor cycle will be used to execute a pair of external instructions.
RUN	The processor is in normal operation with one instruction following another. When the light goes off, the computer stops.
IA	The interrupt control is active (on).
OF	The last data transmission instruction that incremented the word being transmitted increased its value to 2^{16} (this is the Bus Overflow flag).
L	This light displays the contents of the 1-bit link register.

Operating Keys. All of the switches at the lower left except SS and STOP are

interlocked so that they have no effect if RUN is lit. The switches perform these functions when turned on.

START Set all Ready flags, clear all other flags and control flipflops, light FI and RUN, and begin normal operation by executing the instruction at the location specified by SC.

CONT Turn RUN on and begin normal operation in the state indicated by the lights.

READ Display the contents of the memory location addressed by SC in the MB lights, and in the data lights if the thumbwheels are set to 06. Then add 1 to SC. At completion FI is lit.

WRITE Store the contents of the data switches in the memory location specified by SC. Then add 1 to SC. At completion FI is lit and the MB lights display the word stored; the data lights also display the word if the thumbwheels are set to 06.

DISP Display the contents of the source register addressed by the thumbwheels in the data lights. At completion FI is lit.

TRM Transmit the contents of the data switches to the destination register specified by the thumbwheels. At completion FI is lit and the data lights display the word transmitted.

SS This is an alternate-action key. While it is down the processor stops at the end of every cycle it executes. The key is for maintenance purposes and allows the operator to run a diagnostic routine or other program one step at a time. Operations are begun by pressing START, and each succeeding cycle is initiated by pressing CONT.

STOP Stop at the completion of the current instruction with the IR lights displaying the instruction and SC pointing to the next instruction. The control lights at the left indicate the type of cycle the processor will execute when operation is resumed.

This is an alternate-action key, so the operator can run a program one instruction at a time by leaving STOP on, executing the first instruction by pressing START, and executing each succeeding instruction by pressing CONT.

To start the computer one must set the start address of the program in the data switches, set the thumbwheels to 07 (SC), and press TRM to load the address into SC. Then START starts the program at SC. Note that the operator can also continue operations in the current computer state but at any desired location by transmitting a new address to SC before pressing CONT.

Use of the DISP key does not affect the machine state. Thus the operator can stop the computer, examine the contents of any registers, and then continue operation. Use of the TRM key affects only the register selected by the thumbwheels and in some cases a device connected to that register. *Eg* transmitting the data switches to the teletype printer will print the 8-bit character in switches 0-7, thus affecting the state of the Output Ready flag and possibly its interrupt request.

CHAPTER THREE

FUNCTIONAL OPERATORS

These are the optional operators that perform various arithmetic and logical operations, keep track of real time, or just provide general purpose storage. The instructions that control them are simply particular cases of the basic instruction types discussed in §§2.2-2.5.

Execution times are given only for function generating instructions that start an operational sequence in a functional operator. Of course, the actual instruction in the program takes only one cycle, but the whole sequence looks like part of it, since the program cannot continue until the sequence is complete. The time given assumes no interruption. The time that actually elapses from the FO instruction until the result is available is the listed time plus any time used for direct memory access (program interrupts are not allowed).

3.1 BASIC ARITHMETIC AND LOGIC

The arithmetic operator AO contains two registers, AX and AY, both of which can be addressed as source and destination for data. At all times the operator is in some specific functional state such that the operator output, which is addressable as a data source, is the given function of the contents of the two registers. *Eg* turning on system power or starting the processor from the console places AO in the add state, making the output continuously equal to the sum of the numbers AX and AY. Changing the contents of either register changes the output to a new sum. Once in a given state, AO retains that state until changed by the program or by the operator pressing the start key.

The AO output is actually seventeen bits, wherein the extra bit is a carry, or equivalently an extra magnitude bit in a sum. But this extra bit is the overflow of the unsigned addition of AX and AY regardless of the functional state of AO — even if the low order sixteen bits of the AO output are a logical function. The overflow value can be determined only by function testing or reading machine status (the low order sixteen bits of the result are available as data).

The circumstances that generate a carry are obvious when dealing with unsigned numbers. An addition with result greater than $2^{16} - 1$ overflows. In subtraction the condition is the same in terms of adding the twos complement; in terms of the original operands the subtraction $A - B$, which is executed by adding A and the twos complement of B, produces a carry if $A \geq B$. The statement of the carry conditions for signed numbers is more complex, but they are exactly equivalent to the conditions given above if the numbers are simply interpreted as unsigned. In addition, both summands are negative, or their signs differ and their magnitudes are equal or the positive one is the greater in magnitude. In subtraction, say $A - B$, the signs of the operands are the same and $A \geq B$, or the signs differ and A is negative.

Example 3.1

```
        MR      Z,AX
        RR      AX,AY           ;Put word in AX and AY
        MRI     -10,TRP         ;Set up count for eight shifts
        RM      TRP,M2+1
M1:     RS      AY,L1           ;Shift AY into link
        RS      AX,L1           ;Shift link into AX
M2:     MSI     0,P1            ;Count step
        SFM     BOV             ;Done?
        JU      M1              ;No, shift again
        .                       ;Yes, AX has word with bytes swapped
        .
        .
```

The register operator codes are 11 and 12, mnemonics AX and AY respectively. Code 13, mnemonic AO, addresses the arithmetic operator, both for its output and for function generating and testing. The functional state of the operator is available to the program as bits 9 and 8 of the machine status register (in the same configuration as given by this FO instruction).

FOA Function Output, Arithmetic

02	*F*	0 0	13
15 10	9 8	7 6	5 0

Set AO to the state specified by *F* as follows.

Bits 9-8	*Mnemonic*	*Function*
0	ADD	Addition
1	AND	And
2	XOR	Exclusive Or
3	OR	Or

SFA Sense Function, Arithmetic

13	*F*	*N*	02
15 10	9 8 7	6 5	0

Perform a function test (as described in §2.5) on the carry if bit 7 in *F* is 1. The mnemonic AOV places a 1 in bit 7.

The simplest way to determine whether the contents of AX and AY are identical is this:

```
FOA     XOR         ;Exclusive or
JC      AO,ETZ,YES  ;Jump to YES if
                    ;AX = AY
```

The following computes the number ten times that contained in location Z. (Assume Z now has a number less than $2^{16}/10$.)

```
FOA     ADD         ;Add
FOM     CLL         ;Clear link
MR      Z,L1,AX     ;AX = 2Z
RR      AX,L1,AY    ;AY = 4Z
RR      AO,AX       ;AX = 6Z, AO = 10Z
```

Suppose we wish to use the word in location Z with its bytes swapped. Example 3.1 accomplishes this. Note that the example does not use the functional properties of AO; the shifting could just as well be done in a pair of general purpose registers, or in a pair of core locations if not even AO were available (the latter would be longer in both space and time). With both AO and general registers we could keep the count in one of the latter instead of in core; this would eliminate the memory reference in the sixth line (inside the loop) and would eliminate the third line altogether.

Multiply Subroutine. In pencil and paper decimal multiplication, one multiplies the multiplicand by each multiplier digit separately to form a set of partial products. Successive partial products are shifted one place to the left (they are multiplied

Example 3.2

```
MPY:    RMI   TRP,0           ;Save return address
        FOA   ADD             ;Select add
        FOM   CLL             ;Initialize sign flag
        JC    AX,GEZ,MPY1     ;Jump if multiplier positive
        RSC   AX,P1           ;Otherwise negate multiplier
        FOM   STL             ;And set flag
MPY1:   JC    AY,GEZ,MPY2     ;Jump if multiplicand positive
        RSC   AY,P1           ;Otherwise negate multiplicand and complement flag
        FOM   CML             ;(Result will be positive if flag is 0)

MPY2:   RM    AX,MPY3+1       ;Store multiplier in loop
        MRI   -20,AX          ;Set count for 16 steps
        RM    AX,MPY4+1
        ZR    AX              ;Initialize running sum

MPY3:   MSI   0,R1            ;Loop: rotate multiplier (carry sign flag along)
        SFM   NOT LNK         ;Skip if current multiplier bit is 0
        RR    MSR,R1,0        ;Otherwise put AOV in link
        RR    AO,AX           ;And update sum (SF skips two)
        RS    AX,R1           ;Shift sum — put low bit in link for replacing
                              ;multiplier
MPY4:   MSI   0,P1            ;Count step
        SFM   BOV             ;Done?
        JU    MPY3            ;No, store a bit and get another
        MR    MPY3+1,R1,AY    ;Yes, shift last bit into low part, sign flag into
                              ;link; put low half in AY
        SFM   LNK             ;Should product be negative?
        JUD   MPY+1           ;No, return

        RSC   AX              ;Yes, complement high part
        RSC   AY,P1           ;And negate low part
        SFM   NOT BOV         ;Any carry out of low part?
        RS    AX,P1           ;Yes, add it to high part
        NOP
        JUD   MPY+1           ;Return
```

by successive powers of 10) and summed. In the computer it is easier to add each partial product as it is formed and shift the result one place to the right so the running sum is in the correct position to receive the next one. Since the numbers are binary, each partial product is either the multiplicand or zero. Hence at each step we either add the multiplicand and shift or simply shift depending on whether the next bit of the multiplier is 1 or 0.

The multiply subroutine operates on signed numbers in AX and AY to generate a signed double length product whose high and low order parts are left in AX and AY respectively (sign in AX bit 15, 31-bit magnitude in the rest of AX and all of AY). The routine, Example 3.2, is called by a JU MPY.

3.2 GENERAL PURPOSE REGISTERS

These are exactly what the name implies: two nonmemory registers that can be used for any purpose. Their operator codes are 26 and 27, mnemonics GR1 and GR2.

Suppose we wish to exchange the contents

of two memory locations, A and B. If we are limited to the basic processor we must use the sequence on the left, but the addition of general registers allows us to use the shorter sequence on the right.

```
MR   A,TRP          MR   A,GR1
RM   TRP,TEMP       MR   B,GR2
MR   B,TRP          RM   GR2,A
RM   TRP,A          RM   GR1,B
MR   TEMP,TRP
RM   TRP,B
```

If we have both an arithmetic operator and general registers, the multiply subroutine given at the end of the preceding section can be shortened by keeping the multiplier and the step count in GR1 and GR2. The three instructions beginning at MPY2 are replaced by this:

```
MPY2: RR    AX,GR1     ;Or simply start
      MRI   -20,GR2    ;with multiplier
                       ;in GR1
```

This saves three locations and five cycles. But now the MSIs at MPY3 and MPY4 can be replaced by RRs, saving two locations and two cycles in a loop that is iterated sixteen times. Hence the total saving is five locations and thirty-seven cycles.

3.3 BYTE OPERATIONS

The byte handling option card contains two operators, a byte swapper and a byte packer. The former has operator code 24, mnemonic BSW; the latter has code 25, mnemonic BPK.

Sending a word to the swapper makes the same word with its left and right halves interchanged available from the swapper. Suppose AX contains 012345, *ie* the two 8-bit bytes 00010100 and 11100101. This pair of instructions,

```
RR    AX,BSW
RR    BSW,AX
```

changes AX to 162424, *ie* 11100101 00010100.

Addressing the packer as destination in a data transmission instruction causes it to shift bits 0-7 of its own contents into bits 8-15, and accept bits 0-7 of the source register in its own right half. Thus each pair of transfers into BPK packs a pair of bytes from left to right. Say we have the codes for the characters *A* and *B* in bits 0-7 of locations D and D+1, and we wish to pack them with *A* on the left in location C. This suffices.

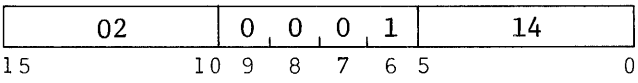
```
MR    D,BPK
MR    D+1,BPK
RM    BPK,C
```

3.4 MULTIPLICATION

Multiplication can be performed much faster than with the multiply subroutine given in §3.1 by adding a multiply operator to the system. The user has a choice of two such operators: one has a read-only memory with a built-in routine that uses the arithmetic operator; the other is purely hardware and is even faster.

The first type has operator code 14, mnemonic MPO. It operates on unsigned integers to generate a double length product whenever the following instruction is given.

```
F0  STRT,MPO          F0, Start Multiplier
32 cycles                          56.32 µs
```



If the arithmetic operator is in the add state, multiply the unsigned integer in AY by the unsigned integer in MPO, and add the product to the unsigned integer in AX. Place the high part of the result in AX, the most significant bit of the low part in the link, and the rest of the low part in MPO bits 15-1. MPO bit 0 retains the original state of the link.

Before using the above instruction the program must set up AO for addition, and it must clear AX if a straight product is desired. Following the multiplication the program should shift MPO right one if the low order part of the product is wanted. This also restores the link, which can be used as a sign flag.

The program must take care of the signs. The usual procedure is as shown in the multiply subroutine already given: use the link for a sign flag, make both operands positive, and then adjust the result. With the operands in AY and MPO, the following sequence performs the unsigned multiplication, with the instruction defined above replacing the loop in Example 3.2.

```
MPY:  RMI   TRP,0       ;Save return
      FOA   ADD         ;address
      ZR    AX
      FO    STRT,MPO    ;Start 32 cycle
                        ;multiply
      RS    MPO,R1      :Finalize low part
                        ;restore link
      JUD   MPY+1       ;Return
```

CHAPTER FOUR

HARDCOPY EQUIPMENT

This chapter discusses the simpler peripheral devices: teletypewriter, tape reader, tape punch, card reader, card punch, plotter and line printer. These devices are used principally for communication between computer and operator using a paper medium: tape, cards, form paper or graph paper. All transfers for them are made by the program.

The program can type out characters on the teletype printer and can read characters that have been typed in at the keyboard. This device has the slowest transfer rate of any, but it provides a convenient means of man-machine interaction. The KSR teletypes comprise only a keyboard and printer; the ASR models also have a slow speed tape reader and punch. This punch and the separate high speed punch supply output in the form of 8-channel perforated paper tape. The information punched in the tape can be brought into the system by the high speed tape reader or the one mounted in the teletype.

The card equipment processes standard 12-row 80-column cards. Many programmers find cards a convenient medium for source program input and for supplying data that varies from one program to another. Cards and paper tape are both convenient to prepare manually, but card input is much faster than tape, and simple changes are easier to make: individual cards can be repunched, and cards can be added or removed from the deck. A possible consideration in using cards is that many installations do not include an online card punch.

The line printer provides text output at a relatively high rate. The program must effectively typeset each line; upon command the printer then prints the entire line. With the plotter, the program can produce ink drawings by controlling the incremental motion of pen on paper in a cartesian coordinate system. Curves and figures of any shape can be generated by proper combinations of motion in x and y.

4.1 TELETYPEWRITER

Two teletype models are regularly available for use with the GRI-909: the ASR33 and KSR33, both of which are capable of speeds up to ten characters per second. The program can type out characters and can read in the characters produced when keys are struck at the keyboard. With an ASR the program can also punch characters in a tape and read characters from a tape.

The teletype separates its input and output functions and is really two distinct devices that share the same operator code. Each device has its own Ready and Interrupt Status flags, and its own interrupt channel and status bit assignments. Placing a code for a character in the output buffer causes the teletype to print the character or perform the designated control function. Striking a key places the code for the associated character in the input buffer where it can be retrieved by the program,

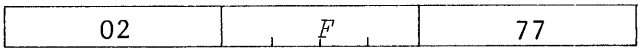
but it does nothing at the teletype unless the program sends the code back as output.

Character codes received from the keyboard have eight bits wherein the most significant is always 1, but the printer ignores this bit in characters transmitted to it (*eg* codes 123 and 323 print the same character). Lower case characters (codes 340-376) are not available on the keyboard, but transmitting a lower case code to the teletype causes it to print the corresponding upper case character. (There are, of course, no restrictions on the codes that can be punched in or read from tape). To go to the beginning of a new line the program must send both a carriage return, which moves the type block to the left margin, and a line feed, which spaces the paper. The horizontal and vertical tabs and form feed have no effect on the printer. Horizontal tabs are usually simulated by spaces, with tab settings at every eighth column (9, 17, ...)

The teletype input and output both use operator code 77, mnemonic TTI or TTO. As the source in a data transmission (or data testing) instruction, this code retrieves a character from the teletype input buffer; as the destination in data transmission, it sends a character to the output buffer. In function generating or testing instructions, it represents both devices.

FO -,TTO FO, Teletype Output

FO -,TTI FO, Teletype Input



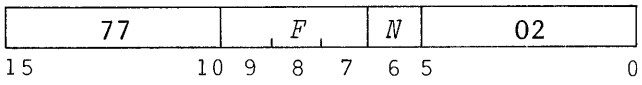
Perform the functions specified by 1s in *F* as follows.

Bit	*Mnemonic*	*Function*
6	STRT	Read one character from tape into the input buffer
7	ORDY	Clear Output Ready
9	IRDY	Clear Input Ready

Programming 1s in bits 6 and 9, mnemonic INP, clears Input Ready and starts the reader.

SF TTO, SF, Teletype Output

SF TTI, SF, Teletype Input



Perform a function test (as described in §2.5) on the flags selected by 1s in *F* as follows.

Bit	*Mnemonic*	*Flag*
7	ORDY	Output Ready
9	IRDY	Input Ready

Teletype Output

Output Interrupt Status is bit 0 of the status register, and the teletype output interrupts to location 11.

Sending a character from bits 0-7 of any source register to the output buffer clears Output Ready (removing the interrupt request) and turns on the transmitter, causing it to send the contents of the output buffer serially to the teletype (the buffer is cleared during transmission). The printer prints the character or performs the indicated control function. If the punch is on, the character is also punched in the tape, with bit 0 corresponding to channel 1 (a 1 produces a hole in the tape). Completion of transmission sets Output Ready, requesting an interrupt if Output Interrupt Status is set.

Timing. The teletype can type or punch up to ten characters per second. After Output Ready is set, the program has 18.18 ms to send another character to keep typing or punching at the maximum rate. The sequence carriage return-line feed, when

iven in that order, allows sufficient time for the type block to get to the beginning of a new line.

Teletype Input

Input Interrupt Status is bit 1 of the status register, and the teletype input interrupts to location 14.

Reception from the keyboard requires no initiating action by the program: striking a key clears Input Ready and transmits the code for the character serially to the input buffer. Completion of reception sets Input Ready, requesting an interrupt if Input Interrupt status is set. Upon retrieving the character in bits 0-7, the program should give an FO IRDY,TTI to clear Input Ready and remove the interrupt request if more input is expected.

If the reader is under program control, giving an FO INP,TTI clears Input Ready (removing the interrupt request) and causes the reader to read all eight channels from the next frame on tape. The reader transmits the frame serially to the buffer, with channel 1 corresponding to bit 0 (the presence of a hole produces a 1 in the buffer). Completion of reception sets Input Ready, requesting an interrupt if Input Interrupt Status is set.

Timing. After Input Ready is set the character is available for retrieval for 20.45 ms before another key strike can destroy it. If the reader is in use, the program has 20.45 ms to give an FO INP,TTI and keep the tape in continuous motion.

Programming Examples

There are basically two procedures for using the function testing instructions in a loop to process a series of characters. Consider this loop for typing out characters from a table beginning at location TAB (we assume the printer is not in use).

```
OUT:  MRID  TAB-1,TTO    ;Type out
      SF    TTO,ORDY     ;Wait till trans-
      JU    .-1          ;mission done
      :                  ;Compute
      :
      JU    OUT          ;Go back
```

This procedure is very poor as most of the time is spent waiting during the transmission, and there is very little time to do anything afterwards if we are to go back to type out the next character at full speed. But with this arrangement:

```
OUT:  SF    TTO,ORDY     ;Wait till printer
      JU    .-1          ;free
      MRID  TAB-1,TTO    ;Type out
      :                  ;Compute, etc
      :
      JU    OUT          ;Go back
```

we have almost all of the time for worthwhile program and we can run at full speed provided only that we jump back to OUT before the entire teletype cycle time is over. Also, the first time into the loop we wait until any previous (perhaps unknown to us) teletype output operation is finished.

Of course, using the interrupt eliminates all waiting time. Suppose we wish to type out twenty characters (one per location) beginning at TAB, using one of the general purpose registers to count the characters. Our main program might set things up like Example 4.1 where we assume that the program left Output Ready on the last time the teletype output was used. Hence an interrupt will occur immediately for the first character. The interrupt routine might be like Example 4.2. If we do not care whether TRP is affected, we could substitute

```
      JC    GR1,ETZ,DONE
```

for testing overflow and loading SC. This saves no time but it takes only two locations instead of three.

Without the interrupt, the dichotomy discussed above exists also for input operations. This is bad:

Example 4.1

```
        MRI   -24,GR1          ;Set up GR1: 20(10) = 24(8)
        MRI   014207,TRP       ;Set up channel locations 7,10
        RM    TRP,7            ;014207 = 06 0010 07 = MRI -,SC
        MRI   OUT-1,TRP
        RM    TRP,10
        FOI   ICO              ;Turn interrupt on
        ZR    P1,ISR           ;Set Output Interrupt Status
        .                      ;Continue program
        .
        .
```

Example 4.2

```
OUT:    MRID  TAB-1,TTO        ;Type out character
        RMI   MSR,0            ;Save machine state
        RS    GR1,P1           ;Count character
        SFM   BOV              ;Done yet?
        MRI   DONE-1,SC        ;Yes
        MR    OUT+3,MSR        ;Restore machine state
        FOI   ICO              ;Turn interrupt back on
        MR    6,SC             ;Return to main program
DONE:   MR    OUT+3,MSR        ;Restore machine state
        ZR    ISR              ;Disable interrupt
        MR    6,SC             ;Return
```

```
IN:     FO    INP,TTO          ;Read character
        SF    TTI,IRDY         ;Wait till recep-
        JU    .-1              ;tion done
        RMID  TTI,TAB-1        ;Store character
        .                      ;Decide whether to
        .                      ;read another,etc
        .
        JU    IN               ;Go back
```

but this is good:

```
IN:     FO    INP,TTI          ;Read character
        .                      ;Lots of time
        .
        .
        SF    TTI,IRDY         ;Wait till recep-
        JU    .-1              ;tion done
        RMID  TTI,TAB-1        ;Store character

        SF    TTO,ORDY         ;Lets make a copy
        JU    .-1              ;of the tape while
        RR    TTI,TTO          ;we are at it

        .                      ;Decide whether to
        .                      ;read another
        .
        JU    IN               ;Do this if want
                               ;another
        .                      ;Skip to here if
        .                      ;not
        .
```

Operation

A KSR is actually two independent devices, keyboard and printer, which can be operated simultaneously. An ASR is really four devices, keyboard, printer, reader and punch, which can be operated in various combinations. Power must be turned on by the operator: the switch is beside the keyboard and is labeled LINE/OFF/LOCAL or ON/OFF and has an unmarked third position opposite ON. When this switch is set to LOCAL or the unmarked position, power is on but the machine is off line and can be used like a typewriter. Moreover, in an ASR, turning on the punch allows the operator to punch a tape from the keyboard, and running the reader allows a tape to control the printer (if the punch is also on, it duplicates the tape).

Turning the switch to LINE or ON connects the unit to the computer and separates its input and output functions. Thus any information transmitted to the computer from the keyboard affects the printer only insofar as the computer sends it back. Turning on the reader places it under program control, and turning on the punch causes it to punch whatever is sent to the printer by the computer.

The only control on the reader is a 3-position switch. When the switch is in the FREE position, the tape can be moved by hand freely through the reader mechanism. The STOP position engages the reader clutch so the tape is stationary but the reader is still off. Turning the switch to START causes the reader to read the tape if the unit is in local, but places it under program control if on line.

The operator controls the punch by means of four pushbuttons. The two on the right turn the punch on and off. Pressing the REL button releases the tape so it can be moved by hand through the punch mechanism. Pressing B.SP. moves the tape backward one frame so the operator can delete a frame that is incorrect by striking the rubout key. Pressing HERE IS with the keyboard in local punches twenty lines of blank tape (lines with only a feed hole punched).

The keyboard resembles that of a standard typewriter. Codes for printable characters on the upper parts of the key tops are transmitted by using the shift key; most control codes require use of the control key. The line feed spaces the paper vertically at six lines to the inch, and must be combined with a return to start a new line. The local line feed and return keys affect the printer directly and do not transmit codes. Appendix F lists the complete teletype code, ASCII characters and key combinations. Pressing the REPT button and striking any character key causes transmission of the corresponding code so long as REPT is held down. Characters that require the shift key may also be repeated in this manner, but there is no repetition of control characters.

Teletype manuals supplied with the equipment give complete, illustrated descriptions of the procedures for loading paper and tape and changing the ribbon. The best and easiest way to learn how to do any of these things is to have someone who knows show you how, but as a precautionary measure we also describe them here.

Tape. The tape moves in the reader from back to front with the feed holes closer to the left edge. To load tape, set the switch to FREE, release the cover guard by opening the latch at the right, place the tape so that the sprocket wheel teeth engage the feed holes, close the cover guard, and set the switch to STOP.

To load tape in the punch, raise the cover, feed the tape manually from the top of the roll into the guide at the back, move the tape through the punch by turning the friction wheel, then close the cover. Turn on the punch with the unit in local and punch about two feet of leader by pressing HERE IS or the control, shift and P keys to generate null codes.

Paper. The printer has an 8½-inch roll of paper at the back. Printed sections can be torn off against the edge of the glass window in front of the platen. To replenish the paper, snap open the cover, remove the old roll and slip a new one in its place. Draw the paper from the roll around the platen as in an ordinary typewriter.

Ribbon. Replace the ribbon whenever it becomes worn or frayed or the printing becomes too light. Disengage the old ribbon from the ribbon guides on either side of the type block, and remove the reels by lifting the spring clips on the reel spindles and pulling the reels off. Remove the old ribbon from one of the reels and replace the empty reel on one side of the machine; install a new reel on the other side. Push down both reel spindle spring clips to secure the reels. Unwind the fresh ribbon from the inside of the supply reel, over the guide roller, through the two guides on either side of the type block, out around the other guide roller,

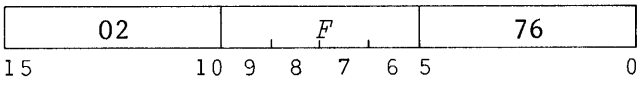
and back onto the inside of the takeup reel. Engage the hook on the end of the ribbon over the point of the arrow in the hub. Wind a few turns of the ribbon to make sure that the reversing eyelet has been wound onto the spool. Make sure the ribbon is seated properly and feeds correctly in operation.

4.2 PAPER TAPE READER AND PUNCH

The high speed reader and punch are totally separate devices — even physically — but they share a single operator code in the same manner that the teletype input and output do. The common operator code is 76, mnemonic HSR or HSP. Each interface contains an 8-bit buffer that corresponds to bits 0-7 of a computer word; the reader buffer is addressable as a source of data, the punch buffer as a destination.

FO -,HSR FO, High Speed Reader

FO -,HSP FO, High Speed Punch



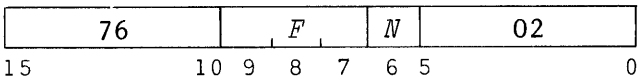
Perform the functions specified by 1s in *F* as follows.

Bit	*Mnemonic*	*Function*
6	STRT	Read one character from tape into the reader buffer
7	ORDY	Clear Punch (Output) Ready
9	IRDY	Clear Reader (Input) Ready

Programming 1s in bits 6 and 9, mnemonic INP, clears Reader Ready and starts the reader.

SF HSR, FO, High Speed Reader

SF HSP, FO, High Speed Punch



Perform a function test (as described in §2.5) on the flags selected by 1s in *F* as follows.

Bit	*Mnemonic*	*Flag*
7	ORDY	Punch Ready
9	IRDY	Reader Ready

Paper Tape Reader

The reader processes 8-channel perforated paper or mylar tape photoelectrically at a speed of 300 frames per second. Reader Interrupt Status is bit 3 of the status register, and the reader interrupts to location 22.

Giving an FO INP,HSR clears Ready (removing the interrupt request) and causes the reader to read all eight channels from the next frame on tape into the buffer, with channel 1 corresponding to bit 0 (the presence of a hole produces a 1 in the buffer). When the operation is complete the reader sets Ready, requesting an interrupt if Interrupt Status is set.

Timing. At 300 frames per second the reader takes 3.3 ms per character, but the program must read several frames before the reader reaches maximum speed. After Ready is set, the program has 1.5 ms to retrieve the character and give an FO INP, HSR to keep the tape in continuous motion. Waiting longer forces the reader to operate at a speed no greater than 150 frames per second.

Operation. Tapes can be oiled or not but must be opaque. To load the reader, place the fanfold tape stack vertically in the bin at the right, oriented so that the front end of the tape is nearer the read head and the feed holes are away from you. Lift the gate, take three or four folds of tape from the bin, and slip the tape into the reader from the front.

Carefully line up the feed holes with the sprocket teeth to avoid damaging the tape, and close the gate. Make sure that the part of the tape in the left bin is placed to correspond to the folds, otherwise it will not stack properly. Turn on the power switch so the reader can respond to the program.

Paper Tape Punch

The punch perforates 8-channel paper tape at speeds up to 60 frames per second. Interrupt Status is bit 2 of the status register, and the punch interrupts to location 17.

Sending a character from bits 0-7 of any source register to the punch buffer clears Ready (removing the interrupt request) and causes the punch to punch the contents of the buffer in the tape, with bit 0 corresponding to channel 1 (a 1 produces a hole in the tape). After punching is complete, the device sets Ready, requesting an interrupt if Interrupt Status is set.

Timing. Punching is synchronized to a punch cycle of 16.7 ms. After Ready sets, the program has 10 ms to send another character to keep punching at the maximum rate; after 10 ms punching is delayed until the next cycle.

Example. With direct function processing a program for duplicating a tape is quite simple.

```
DUP:  FO    INP,HSR     ;Read
      SF    HSR,IRDY    ;Wait for character
      JU    .-1
      SF    HSP,ORDY    ;Got it, wait for
      JU    .-1         ;punch
      RR    HSR,HSP     ;Move character to
                        ;punch
      JU    DUP         ;Read another
```

Operation. Punch power must be left on all the time that the punch might be used as it otherwise will not respond to the program. Fanfold tape is fed from a box behind the punch inside its enclosure. After it is punched, the tape moves into a storage bin from which the operator may remove it through a slot in the front. Pushing the feed button beside the slot clears the buffer and punches blank tape (tape with only feed holes punched) as long as it is held in, provided power is on.

To load tape, first empty the chad box. Then tear off the top of a box of fanfold tape (the top has a single flap; the bottom of the box has a small flap in the center as well as the flap that extends the full length of the box). Set the box in the frame and thread the tape through the punch mechanism. The arrows on the tape should be on top and should point in the direction of tape motion. If they are underneath, turn the box around. If they point in the opposite direction, the box was opened at the wrong end; remove the box, seal up the bottom, open the top, and thread the tape correctly.

To facilitate loading, tear or cut the end of the tape diagonally. Thread the tape under the out-of-tape plate, open the guide plate (over the sprocket wheel), push the tape beyond the sprocket wheel, and close the guide plate. Press the feed button long enough to punch about a foot and a half of leader. Make sure the tape is feeding and folding properly in the storage bin.

To remove a length of perforated tape from the bin, first press the feed button long enough to provide an adequate trailer at the end of the tape (and also leader at the beginning of the next length of tape). Remove the tape from the bin and tear it off at a fold within the area in which only feed holes are punched. Make sure that the tape left in the bin is stacked to correspond to the folds; otherwise, it will not stack properly as it is being punched. After removal, turn the tape stack over so the beginning of the tape is on top, and *label it* with *name*, *date*, and other appropriate information.

4.3 BOOTSTRAP LOADERS

Before a program can be executed it must be brought into memory. This requires

Example 4.3

```
;BOOTSTRAP LOADER: TELETYPE, BYTE PACKER

        FOM   HLT              ;Halt between blocks
BLTP:   FO    INP,TTI          ;Get first (control) frame
        SF    TTI,IRDY
        JU    .-1
        JC    TTI,ETZ,BLTP-1   ;Jump if end of block

        FO    INP,TTI          ;Got control frame — ignore it and get first byte
        SF    TTI,IRDY         ;(second frame)
        JU    .-1
        RR    TTI,BPK          ;Left byte (8-15) to packer
        FO    INP,TTI          ;Get second byte (third frame)
        SF    TTI,IRDY
        JU    .-1
        RR    TTI,BPK          ;Right byte (0-7) to packer

M1:     RMID  BPK,0            ;Store target word
        JU    BLTP             ;Continue
```

Example 4.4

```
;BOOTSTRAP LOADER: TELETYPE, ARITHMETIC OPERATOR

        FOM   HLT              ;Halt between blocks
BLTA:   FO    INP,TTI          ;Get first (control) frame
        SF    TTI,IRDY
        JU    .-1
        JC    TTI,ETZ,BLTA-1   ;Jump if end of block

        RM    TTI,M2+1         ;Store control frame for shift count
        FO    INP,TTI          ;Get first byte (second frame)
        SF    TTI,IRDY
        JU    .-1
        RR    TTI,AX           ;Left byte (8-15) to AX (0-7)
        FO    INP,TTI          ;Get second byte (third frame)
        SF    TTI,IRDY
        JU    .-1
        RR    TTI,AY           ;Right byte (0-7) to AY

        FOM   CLL              ;Initialize link
M1:     RS    AX,L1            ;Loop, shift AX left 8
M2:     MSI   0,R1
        SFM   LNK              ;Has control frame set link?
        JU    M1               ;No, shift again

M3:     RMID  AO,0             ;Yes, store target word
        JU    BLTA             ;Continue
```

that a loading program already reside in core. If the memory is empty, one can use the console switches to load in a bootstrap loader, which is ordinarily used only to bring in a more extensive block loader. This latter program is then used to read the object tapes of all other programs. Both the bootstrap and the block loader usually reside in high core where they are not disturbed by any of the standard GRI-909 software. But if an undebugged user routine inadvertently destroys the block loader, it can be restored by first reloading the bootstrap

manually.

There are several bootstrap loaders depending on which functional operators are included in the system, and for each there are two versions, one for the teletype reader, the other for the high speed reader. Every time any bootstrap loader is used, the operator must key in an address one less than the first location that is to be loaded.

Every bootstrap loader reads a tape in a special format in which each word requires three frames. The first is the control code 200, the second and third are the left and right 8-bit bytes of the binary word to be stored. A block may contain any number of 3-frame sets, but it is recommended that they be kept short. The tape should begin with blank frames (*ie* leader), and null codes separate the blocks. The loader halts every time it encounters a block separator (*ie* a null frame that is not in a 3-frame program segment), but it can be restarted simply by pressing the continue key.

Example 4.3 is a bootstrap loader that utilizes the teletype and the byte packer. To use it the operator must key the initial load address minus one into location M1+1 and start the bootstrap from the console at location BLTP. Example 4.4 uses the arithmetic operator instead of the byte packer. One less than the load address must be keyed into location M3+1. The loader uses the control frame (200) for the shift count, and since it is started by the start key, AO is set up for addition.

The table on the next page lists the memory words for the two loaders given above and also for one that uses no optional functional operators. All are written for the teletype reader; for the high speed reader simply substitute operator code 76 wherever 77 appears in the list. Always place the loader in the very top of core. Thus to key in the loader for the arithmetic operator in an 8K memory, first set 017742 in the switch register, set the thumbwheels to 07, and press TRM. Then successively set each word in the switch register and press WRITE.

To use the bootstrap to load the block loader or any other program in the special format, follow these steps:

1. Put the special format tape in the reader and turn it on.
2. Set the address of the location that must contain the initial load address - 1 in the switch register (address *xx*775 for the arithmetic operator or byte packer, otherwise *xx*770).
3. Set the thumbwheels to 07 and press TRM.
4. Set the initial load address - 1 in the switch register and press WRITE.
5. Set the start address of the bootstrap (the second address from the top in the appropriate column of the table) in the switch register and press TRM.
6. Press START

The bootstrap will halt at every block separator and following the final block with the start address of the bootstrap in SC.

BOOTSTRAP LOADERS

Basic Processor Only

```
xx727   02 0100 00
xx730   02 1001 77
xx731   77 1000 02
xx732   00 0100 03
xx733   0xx731
xx734   77 0100 03
xx735   0xx727
xx736   77 0000 06
xx737   0xx761
xx740   77 0000 06
xx741   0xx772
xx742   02 1001 77
xx743   77 1000 02
xx744   00 0100 03
xx745   0xx743
xx746   77 0001 06
xx747   0xx770
xx750   02 1001 77
xx751   77 1000 02
xx752   00 0100 03
xx753   0xx751
xx754   77 0000 06
xx755   0xx757
xx756   06 1010 06
xx757   0
xx760   06 1110 06
xx761   0
xx762   00 0100 02
xx763   00 0100 03
xx764   0xx756
xx765   06 1000 06
xx766   0xx757
xx767   06 1000 06
xx770
xx771   06 1110 06
xx772   0
xx773   00 0100 02
xx774   00 0100 03
xx775   0xx765
xx776   00 0100 03
xx777   0xx730
```

Key load address - 1 into xx770. Start at xx730.

Arithmetic Operator

```
xx742   02 0100 00
xx743   02 1001 77
xx744   77 1000 02
xx745   00 0100 03
xx746   0xx744
xx747   77 0100 03
xx750   0xx742
xx751   77 0000 06
xx752   0xx770
xx753   02 1001 77
xx754   77 1000 02
xx755   00 0100 03
xx756   0xx754
xx757   77 0000 11
xx760   02 1001 77
xx761   77 1000 02
xx762   00 0100 03
xx763   0xx761
xx764   77 0000 12
xx765   02 0001 00
xx766   11 1000 11
xx767   06 1110 06
xx770   0
xx771   00 0100 02
xx772   00 0100 03
xx773   0xx766
xx774   13 0011 06
xx775
xx776   00 0100 03
xx777   0xx743
```

Key load address - 1 into xx775. Start at xx743.

Byte Packer

```
xx753   02 0100 00
xx754   02 1001 77
xx755   77 1000 02
xx756   00 0100 03
xx757   0xx755
xx760   77 0100 03
xx761   0xx753
xx762   02 1001 77
xx763   77 1000 02
xx764   00 0100 03
xx765   0xx763
xx766   77 0000 25
xx767   02 1001 77
xx770   77 1000 02
xx771   00 0100 03
xx772   0xx770
xx773   77 0000 25
xx774   25 0011 06
xx775
xx776   00 0100 03
xx777   0xx754
```

Key load address - 1 into xx775. Start at xx754.

Note: For the high speed reader substitute operator code 76 for every 77 that appears above.

APPENDIX A

THE SYSTEM ORIENTED ASSEMBLY LANGUAGE, FAST

The body of this manual describes the instructions that can be performed by the GRI-909 and gives the mnemonic terms employed to write a program using the basic assembly language, BASE. But there is another assembler that allows the programmer to write in a functional or system-oriented language. The end result of either language is the same: the 16-bit words produced for the object program are identical, since the absolute instruction format is a function of the hardware. But in place of the terse symbology of the basic assembly language, FAST has a vocabulary and a grammar that allow one to write a program using statements that are similar to those in ordinary English.

In order to program properly, the programmer is strongly advised to read the body of this manual, particularly Chapters 1 and 2, not only to learn how to handle the various operators, input-output, interrupt programming and the like, but also to learn precisely what each instruction does. But if the programmer is going to use FAST, then he can ignore the basic instruction mnemonics defined for BASE and can skip the discussion of the various programming conventions for that language.

To a great extent the two languages actually share a common vocabulary, and most of the terms defined in the body of the manual have the same meaning (and hence the same numerical value) in FAST. The primary difference between the two languages is in the way the terms are used and the fact that FAST has additional operational terms that produce the similarity to ordinary English. The basic mnemonic terms for operator codes, output functions, test conditions, etc defined for FAST are the following.

Operator Codes (Device Addresses)

AO	Arithmetic Operator
AX	Register AX
AY	Register AY
BPK	Byte packer
BSW	Byte swapper
GR1	General register 1
GR2	General register 2
HSR	High speed paper tape reader
HSP	High speed paper tape punch
ISR	Interrupt status register
MPO	Multiply operator
MSR	Machine status register
SUM	Adder
SWR	Console switch register
SC	Sequence counter
TRP	Trap register
TTI	Teletype input
TTO	Teletype output
ZERO	Null (0) operator

Data Transmission Terms

C	Complement of source
I	Immediate addressing
D	Deferred addressing
.	The current address
P1	Increment by 1
L1	Shift left one bit
R1	Shift right one bit

Data Test Conditions

ETZ	Equal to zero
LTZ	Less than zero
GTZ	Greater than zero
NEZ	Not equal to zero
GEZ	Greater than or equal to zero
LEZ	Less than or equal to zero

Functions (Output Pulses)

ADD	Select AO addition
AND	Select AO AND function
OR	Select AO OR function
XOR	Select AO exclusive OR function
CLL	Clear link
STL	Set link
CML	Complement link
HLT	Halt machine
CLIF	Clear input (Ready) flag (TTI, HSR)

CLOF	Clear output (Ready) flag (TTO, HSP)
ICF	Interrupt control off
ICO	Interrupt control on
STRT	Start

Function Test Conditions

AOV	Arithmetic Overflow flag
BOV	Bus Overflow flag
SOV	Sum overflow flag
LNK	Link
IRDY	Input Ready (TTI, HSR)
ORDY	Output Ready (TTO, HSP)
POK	Power ok
NOT	Negate test condition

Other basic programming conventions given on page 1-10 are also used by FAST. The period represents the current address, a colon following a symbol indicates that it is a symbolic location name, and anything written at the right of a semicolon is commentary that explains the program but is not part of it.

The system language also contains these operational symbols.

TO	Move data or control information from the specified source to the specified destination.
IF ... GO TO	Test data from the specified source and transfer program control (*ie* jump) to the specified location if the test condition is satisfied.
GO TO	Transfer program control (jump) to the specified location.
SKIP (IF)	Test a function (status) and skip the next two locations if the test is positive.

By combining these operational terms with the operator codes and other mnemonic symbols that are mostly shared with BASE, instructions in a program can be represented by statements that are closer to those of a natural language. Moreover the structure of the language actually allows the programmer to use the symbols for operators, test conditions, and the like, in a more flexible way than they can be used in the basic assembler.

Data Transmission

Nonmemory reference data transmission instructions are of the form

Register X (Modification) TO *Register Y*

or more generally,

Source (Modification) TO *Destination*

where neither the source nor the destination is memory. Hence to take a word from AX, increment it by one, and place the result in AY, give

```
AX P1 TO AY
```

To use the complement of the source, place the letter C in front of the statement. Thus

```
C AX P1 TO AY
```

places the twos complement of AX in AY. The twos complement of AX is placed in AX (in other words the register is transferred to itself) simply by failing to specify any destination (the TO is also dropped):

```
C AX P1
```

Data transmission instructions that reference memory are of either of these two forms:

Register (Modification) TO *Location*

Location (Modification) TO *Register*

The letters I and D, for immediate and deferred addressing, precede the term (if any) that specifies the location. Thus to store AX in location ANS, give

```
AX TO ANS
```

but to store it in the location following the current instruction, give

```
AX TO I
```

To load AX from ANS give

```
ANS TO AX
```

but to load AX from the location one greater than that addressed by the contents of say POINT, give

D POINT TO AX

For either type of data transmission instruction, ZERO specifies a null operator. This supplies zero when specified as a source, and as a destination it allows the programmer to affect Bus Overflow or the link without affecting any operator. Thus

ZERO TO AX

clears AX, whereas

GR1 P1 TO ZERO

sets Bus Overflow if GR1 contains $2^{16}-1$.

The following are typical examples of data transmission instructions in FAST.

TTI TO TTO	Send a character from teletype input to teletype output
AX L1	Shift AX left one bit (equivalent to AX L1 TO AX)
ZERO P1 TO AX	Set AX to +1
C ZERO TO AX	Set AX to -1
AX L1 TO ZERO	Set the link if AX is negative (bit 15 is 1)
AX R1 TO ZERO	Set the link if AX is odd
GR1 P1 TO Z+1	Increment the word from GR1 and store it in location Z+1
I 21 TO AX	Load the number 21 into AX
COUNT P1 TO GR1	Increment the word from location COUNT and place the result in GR1
0 TO SC	Load the contents of location 0 into SC (this is the return from a standard interrupt routine)
COUNT P1	Increment the word in location COUNT
I 0 P1	Increment the contents of the location following the current instruction
X L1	Shift the contents of location X left one bit

Data Testing

These instructions have the form

IF *Operator Condition* GO TO *Location*

Deferred addressing is selected by a D preceding the location. Thus the statement

IF TTI ETZ GO TO AGAIN

jumps to location AGAIN if the character in the teletype input buffer is zero.

IF ISR NEZ GO TO D EXIT

jumps to the location whose address is one greater than the contents of location EXIT if any device is enabled to interrupt. To jump unconditionally to location EXIT simply give

GO TO EXIT

Function Generation

The form for these instructions is

Function TO *Operator*

Thus the statement

STRT TO HSR

causes the high speed paper tape reader to read the next character on tape,

ADD TO AO

sets up the arithmetic operator for addition. Functions for the same destination may be combined by giving them together, *eg*

CLIF STRT TO TTO

clears the Input Ready flag and causes the teletype operator to read the next character from tape. The structure of FAST allows the mnemonic term for a function to be given alone if it represents a function that applies to only one destination operator. Hence the following are complete function generating instruction statements in FAST.

CLL Clear link

STL	Set link
HLT	Halt
OR	Select OR function in AO
ADD	Select addition in AO
CML	Complement link (= CLL STL)

Function Testing

The form for these statements is

SKIP IF *Operator Function*

where IF is optional and need not be given. In a manner analogous to function generating instructions, the source operator need not be given if the mnemonic for the function is unique to that operator. Hence the statement

```
SKIP IF TTI IRDY
```

skips the next two locations if teletype input is ready, whereas the simpler statement

```
SKIP IF BOV
```

tests Bus Overflow for a skip. Since the IF is optional, statements can be made even simpler; *eg*

```
SKIP AOV
```

tests Arithmetic Overflow.

An instruction can test whether any of several conditions is true for the same operator by giving the mnemonics together, and the test conditions can be negated, *ie* the test can be that none of the named conditions are true by placing NOT before the terms that specify the functions to be tested. Thus

```
SKIP BOV LNK
```

skips if either Bus Overflow or the link is set, whereas

```
SKIP NOT BOV LNK
```

skips if neither Bus Overflow nor the link is set.

Sample Programs

To better show the relation between the two assembly languages the following routine written in FAST is the same program as Example 2.2 on page 2-9.

```
SUB:    TRP TO I
NEXT:   D SUB+1 TO TRP
        IF TRP ETZ GO TO END
        .
        .
        .
        GO TO NEXT
END:    .
        .
        .
        GO TO D SUB+1
```

On page 2-12 is a sample program that forms a twos complement without using a general register. In FAST it would look like this.

```
        I -21 TO TRP
        TRP TO M2+1
M1:     Z R1
        CML
M2:     I 0 P1
        SKIP BOV
        GO TO M1
        Z P1
```

Example 3.2 is a multiply subroutine that uses the arithmetic operator but does not use general purpose registers. This is the same program in FAST.

```
MPY:    TRP TO I
        ADD
        CLL
        IF AX GEZ GO TO MPY1
        C AX P1
        STL
MPY1:   IF AY GEZ GO TO MPY2
        C AY P1
        CML

MPY2:   AX TO MPY3+1
        I -20 TO AX
        AX TO MPY4+1
        ZERO TO AX

MPY3:   I 0 R1
        SKIP NOT LNK
        MSR R1 TO ZERO
        AO TO AX
        AX R1

MPY4:   I 0 P1
        SKIP BOV
        GO TO MPY3
        MPY3+1 R1 TO AY
        SKIP LNK
        GO TO D MPY+1
```

```
C AX
C AY P1
SKIP NOT BOV
AX P1
NOP
GO TO D MPY+1
```

APPENDIX B

INTERFACING

Functional operators and peripheral device operators of the user's own design can be added to the GRI-909 bus system with great ease. The addition of a source register makes its data available to all destination operators (including memory and the data tester) through the bus modifier. Similarly a new destination register can receive data from any source already in the system, again through the bus modifier. Control over any new operator is exercised by the function generating instructions, which operate it, and the function testing instructions, which determine its state.

Chapter 2 describes all of the GRI-909 instruction types and in particular explains the use of the instructions to control functional and device operators. The reader should be very familiar with the contents of Chapter 2 before he attempts to interface operators of his own design.

Operators are often referred to as devices, especially in engineering drawings and other hardware oriented documents. SDA and DDA stand for "source device address" and "destination device address".

I PHYSICAL ARCHITECTURE

Figure 1 shows the physical organization of the basic package. The frame work is of extruded anodized aluminum serving both as a caged grounding scheme and as guides for the printed circuit cards in the system. The console is enclosed in the hinged door on the front of the cabinet. Mounted on the door are all of the console switches and indicators with their driving and sensing circuits.

All back panel wiring inside the frame is on printed circuit cards with PC card sockets soldered onto them. Thus one card is joined directly to another through plugs, and no wire wrapping or point-to-point wiring is used anywhere in the system. The processor bus has connectors for nine 9×13-inch plug-in cards, three of which are available for large firmware options such as the arithmetic operator. The IO bus has sixteen positions for 9×4-inch cards for smaller firmware or device operators. The memory bus is used for core memory modules and generally is not of interest to the interface engineer.

All connections between the buses and the console are made with multiconductor flat cable attached at both ends to PC cards. Expansion chassis to extend the memory bus and the IO bus are of the same type of construction as the main frame. These expanders are placed either above or below with flat cable connecting the buses.

II INTERFACE PC CARDS

Interfaces are of two general types, and all are referred to as operators. A system interface that is not associated with some external device is a functional or firmware operator. Typical firmware operators are those for basic arithmetic, multiplication, square root, etc. Operators of this type are actually extensions of the processor. Device operators are interfaces to mate the bus system to some external device such as an A-D converter.

Device operators can be built on one or more IO interface cards (Figure 2). This card can contain 33 integrated circuit packages, and has provision for connecting to both the source and destination IO buses and to an external device. The external printed circuit connector has 48 pins and mates to an Amphenol connector type 583167-1. Contacts and keys for those connectors are purchased separately and only those needed are used.

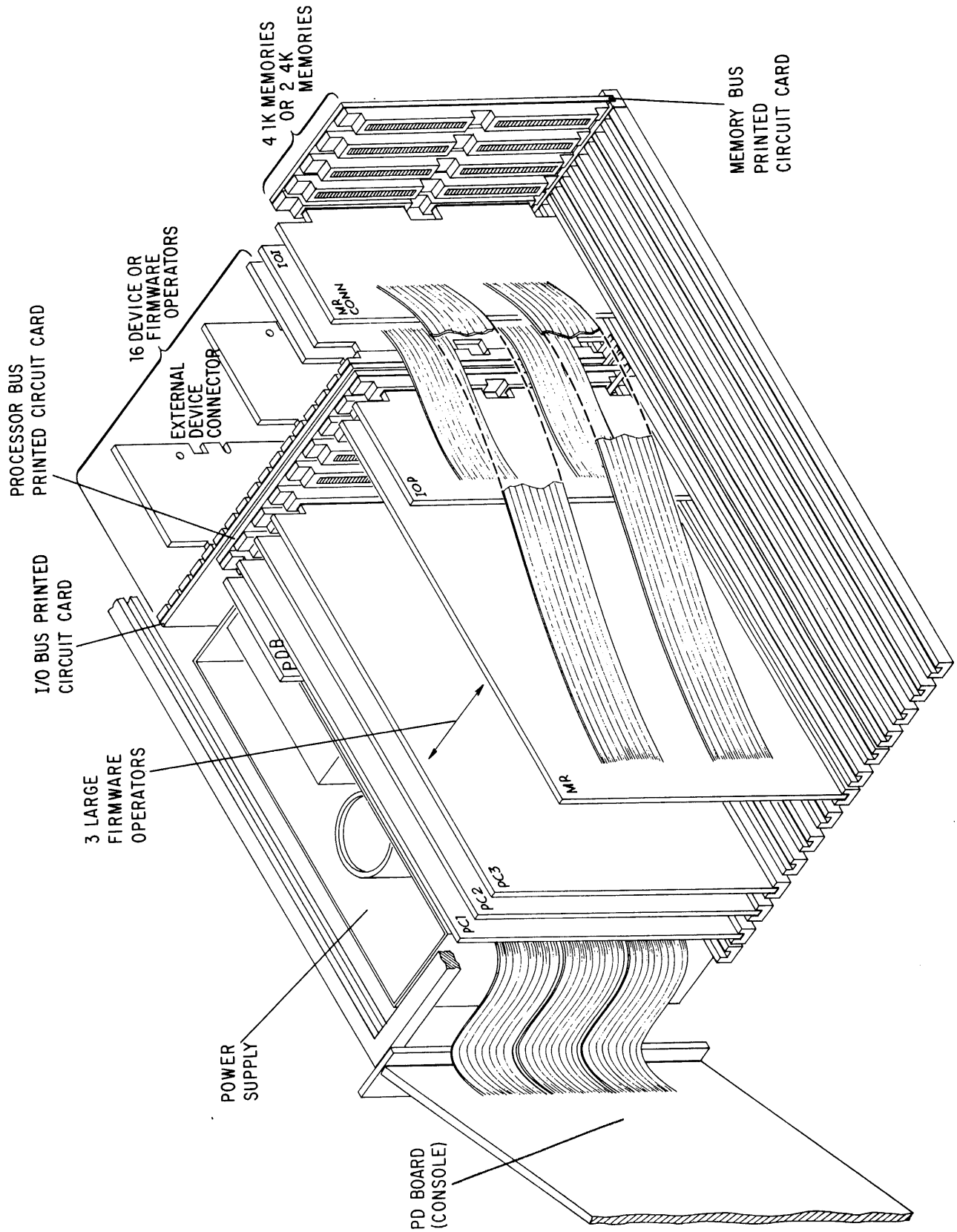


Figure 1. Physical Organization

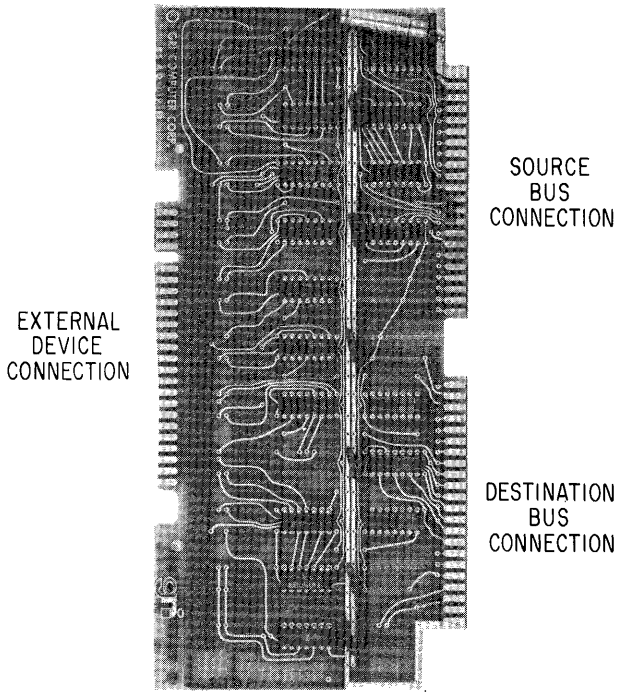
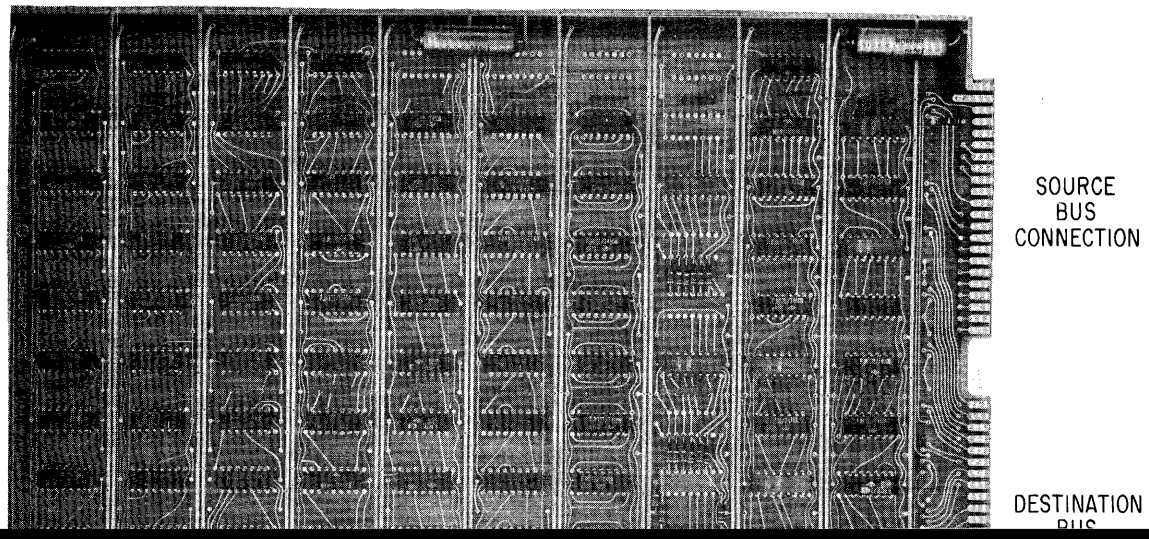


Figure 2. Small Firmware/Device Operator Card (Component Side)



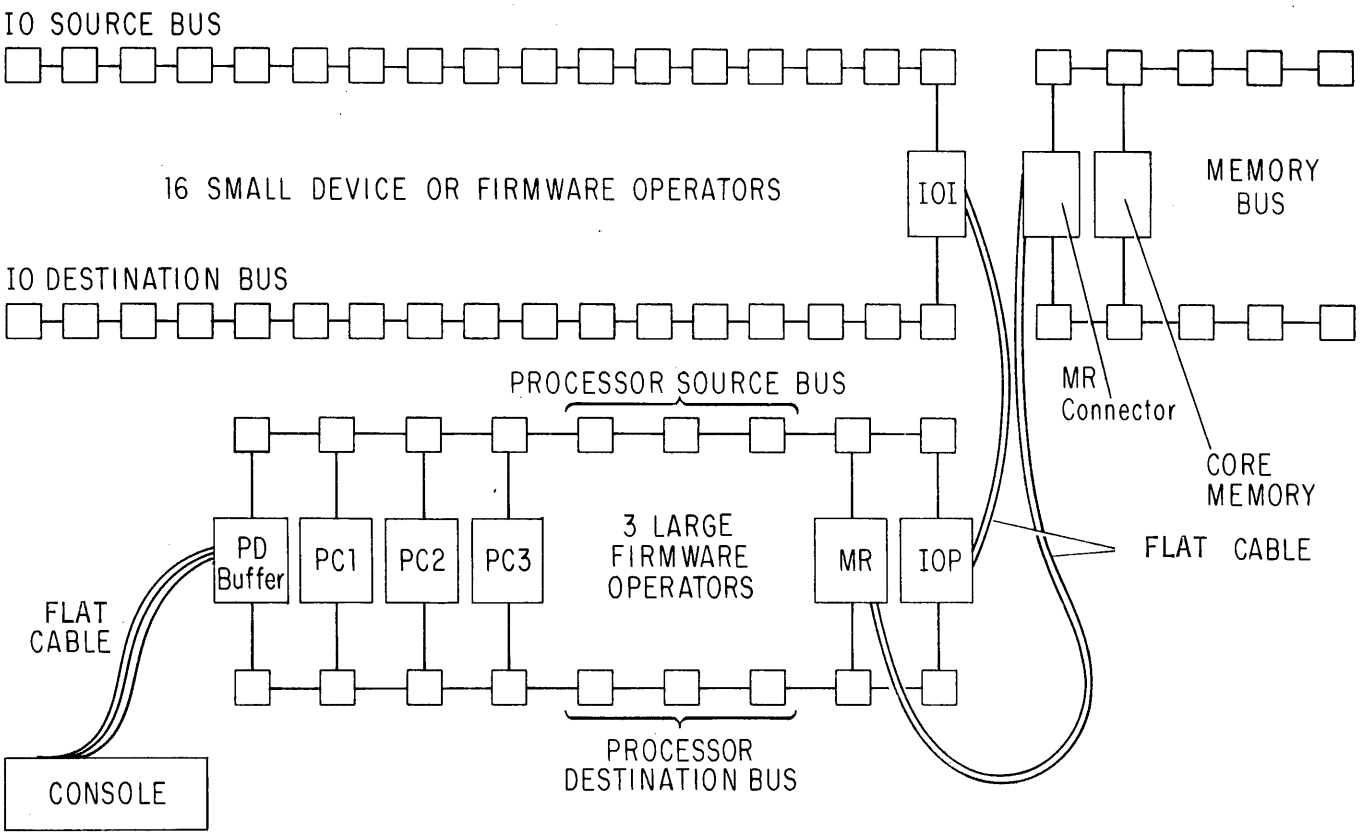


Figure 4. System Busing

Firmware operators may be built on these cards (without using the external device connectors), but one or more such operators may be built on the larger card shown in Figure 3. This card holds up to 108 ICs.

III SYSTEM BUSING

The busing scheme and connections internal to the main chassis are shown in Figure 4. All PC cards shown in the lower row are of the larger size (9×13 inches). The processor is contained on three large cards, labeled PC1, PC2 and PC3. The PD buffer board provides the buffering and cable connection to the console. The MR board contains the MA and MB registers, memory controls, and a connection via cable to the memory bus. Three connectors are available in the processor bus for the addition of firmware operators.

The IOP and IOI cards have flat cable between them to connect the processor bus to the IO bus. Drive circuits on these cards provide signal isolation between the two buses, and decoders provide operator code signals to the IO bus that are not available on the processor bus. These signals are used for interrupt and direct memory access control and are generally not needed by firmware operators.

Both the processor bus and the IO bus are really each two buses, source and destination. In general the destination bus is associated with output from system operators, whereas the source bus is the source of data and control information for input to system operators. The tables

below describe all bus signals and show the pin connections for them (an asterisk indicates a signal that appears only on the IO bus). All signals have two states, low and high, which correspond to nominal voltage levels of 0 and +4 volts. The last letter of every signal name is either H (for high) or L (for low); the level so indicated is the voltage level on the line when the signal represents a 1 or produces the listed function. The plug-in side of the connectors are shown.

Source Bus

Ground	1	A	Ground
+5 V	2	B	+5 V
CLRH	3	C	EIH
DMH	4	D	BKH
PINL	5	E	POUTL
DINL	6	F	DOUTL
DAB1H	7	H	DAB0H
DAB3H	8	J	DAB2H
DAB5H	9	K	DAB4H
DSTRH	10	L	XCLL
CLIBH	11	M	INTBL
IMBL	12	N	DIRBL
STPKL	13	P	STKL
EIRL	14	R	DMRL
SBO1H	15	S	SBO0H
SBO3H	16	T	SBO2H
SBO5H	17	U	SBO4H
SBO7H	18	V	SBO6H
SBO9H	19	W	SBO8H
SB11H	20	X	SB10H
SB13H	21	Y	SB12H
SB15H	22	Z	SB14H

Destination Bus

DBO1L	1	A	DBO0L
DBO3L	2	B	DBO2L
DBO5L	3	C	DBO4L
DBO7L	4	D	DBO6L
DBO9L	5	E	DBO8L
DB11L	6	F	DB10L
DB13L	7	H	DB12L
DB15L	8	J	DB14L
SABOH	9	K	ISYNH
SAB1H	10	L	FTB1L
SAB2H	11	M	FTB2L
SAB3H	12	N	FTB3L
SAB4H	13	P	LINKH
SAB5H	14	R	BOH
IDAH*	15	S	P2H
EASH*	16	T	ISAH*
EDDH*	17	U	EDSH*
EXTH	18	V	FUNCH
CB3H	19	W	CB2H
CB1H	20	X	CBOH
-A	21	Y	+A
Ground	22	Z	Ground

Source Bus

EIH	Out	External Instruction: the processor external instruction cycle.
CLRH	Out	Clear: system clear level generated by the start key or power on or off.
BKH	Out	Break: the processor interrupt cycle.
DMH	Out	Direct Memory: the processor direct memory access cycle.
POUTL	Out	Priority Out: the serial interrupt priority-determining level out of an operator.
PINL	Out	Priority In: the serial interrupt priority-determining level into an operator.
DOUTL	Out	DMA Out: the serial DMA priority-determining level out of an operator.
DINL	Out	DMA In: the serial DMA priority-determining level into an operator.

DAB0H-DAB5H	Out	Destination Address Bus: the 6-bit address of the destination operator.
XCLL	Out	Crystal Clock: a square wave with period 110 ns.
DSTRH	Out	Data Strobe: used by an operator to gate in data from the source bus.
INTBL	In	Interrupt Bus: a common line for all operators to request an interrupt.
CLIBH	Out	Clear Interrupt Bus: clears the DMA Device Service flipflop in the requesting device.
DIRBL	In	Direction Bus: indicates data transfer direction for DMA (0 volts = out, +4 volts = in).
IMBL	In	Increment Memory Bus: increment the memory location during a DMA cycle.
STKL	In	Start Key: signal generated by the start key.
STPKL	In	Stop Key: signal generated by the stop key.
DMRL	In	Direct Memory Access Request: a common line for all operators to request DMA.
EIRL	In	External Instruction Request: a common line for all operators to request an external instruction cycle.
SB00H-SB15H	Out	Source Bus Data: the 16 data lines in the source bus.

Destination Bus

DB00L-DB15L	In	Destination Bus: the 16 data lines in the destination bus.
ISYNH	Out	Interrupt Sync: generated in each cycle to synchronize interrupt and DMA requests by operators.
SAB0H-SAB5H	Out	Source Address Bus: the 6-bit address of the source operator.
FTB1L-FTB3L	In	Function Test Bus: operators use these lines for status input during an SF instruction.
LINKH	Out	Link: the link bit associated with the bus modifier.
BOH	Out	Bus Overflow: the Bus Overflow flag associated with the bus modifier.
P2H	Out	Time 2 Pulse: strobe that gates FO commands into an operator.
IDAH	Out	Interrupt Destination Address: the destination address is 04, the interrupt status register.
ISAH	Out	Interrupt Source Address: the source address is 04, the interrupt status register.
EASH	Out	External Address, Source: the source address is 16, the DMA address register or interrupt address generator.
EDSH	Out	External Data, Source: the source address is 15, the DMA data register or logic that supplies an external instruction from a read-only memory.

EDDH	Out	External Data, Destination: the destination address is 15, the DMA data register.
FUNCH	Out	Function: the current processor instruction is an SF or an FO.
EXTH	Out	Execute Time. A T2 period when programmed transfer occurs.
CBOH-CB3H	Out	Control Bus: lines that transmit control bits in an FO instruction.
+A, -A	Out	Unregulated voltage (26-35 volts) for use in local regulators.

IV INTERFACE LOGIC AND TIMING

There are two types of in-out data transfer: the movement of words or characters by the program and the automatic transfer of data via direct memory access. The program can handle in-out by sensing Ready or by allowing the device to interrupt when it requires service. If the device is automatic, it can use direct memory access for the transfer of data and require response by the program only for control purposes (*eg* when a block transfer is complete or there is some special situation, such as an error, which the program must handle). An optional operator, particularly a firmware operator, may also carry out operations by means of external instructions.

There are thus six categories of functions used in operating equipment added to the system: programmed data transfers, function generation, function testing, interrupts, direct memory access, and external instructions. Typical circuit configurations and timing for these are given here. The timing diagrams show the relationships among the various bus signals involved in the operations. Each line for a control signal represents the actual voltage level. For groups of signals that carry binary information, such as data and addresses, a raised section in the line indicates the time during which that information is held on the bus.

Programmed Data Transfers

Figure 5 shows the timing and hardware for data transmission between an interface register and the bus system. If a register is to receive data it must be connected to the source bus data lines SB00H to SB15H and its input gating must include a decoding net for the destination address lines DAB0H to DAB5H so that only this operator responds when its code is specified as the destination.

It is recommended that the data lines be connected to the D inputs of type 7474 flipflops; these offer the greatest versatility in that they also have dc clear and set inputs for external data entry. The clock input to the data register is derived by combining the data strobe DSTRH with the output of the address decoder. As can be seen in the timing diagram, the strobe occurs at the end of the interval in which the destination address and source data are both valid. Since the address lines carry high levels, 1s can be recognized by connecting the lines directly to the inputs of a decoder such as the 7430 gate, whereas lines for 0s must be connected through inverters. *Eg* to decode address 75, DAB1H is connected through an inverter, the remaining lines are connected directly. Should a master clear signal be desired for the data register, the CLRH line is available; this carries +5 volts during the power-up and power-down sequences and also every time a start signal is sent from the console or remotely. CLRH is normally low and must therefore be inverted to drive the direct clear inputs of the data register.

If a register being added is to supply data to the system, its outputs must be connected through open collector gates, such as the 7401, to the destination bus data lines DB00L to DB15L. The gating input for the register output into the 7401s is derived by decoding the address that appears on source address lines SA8OH

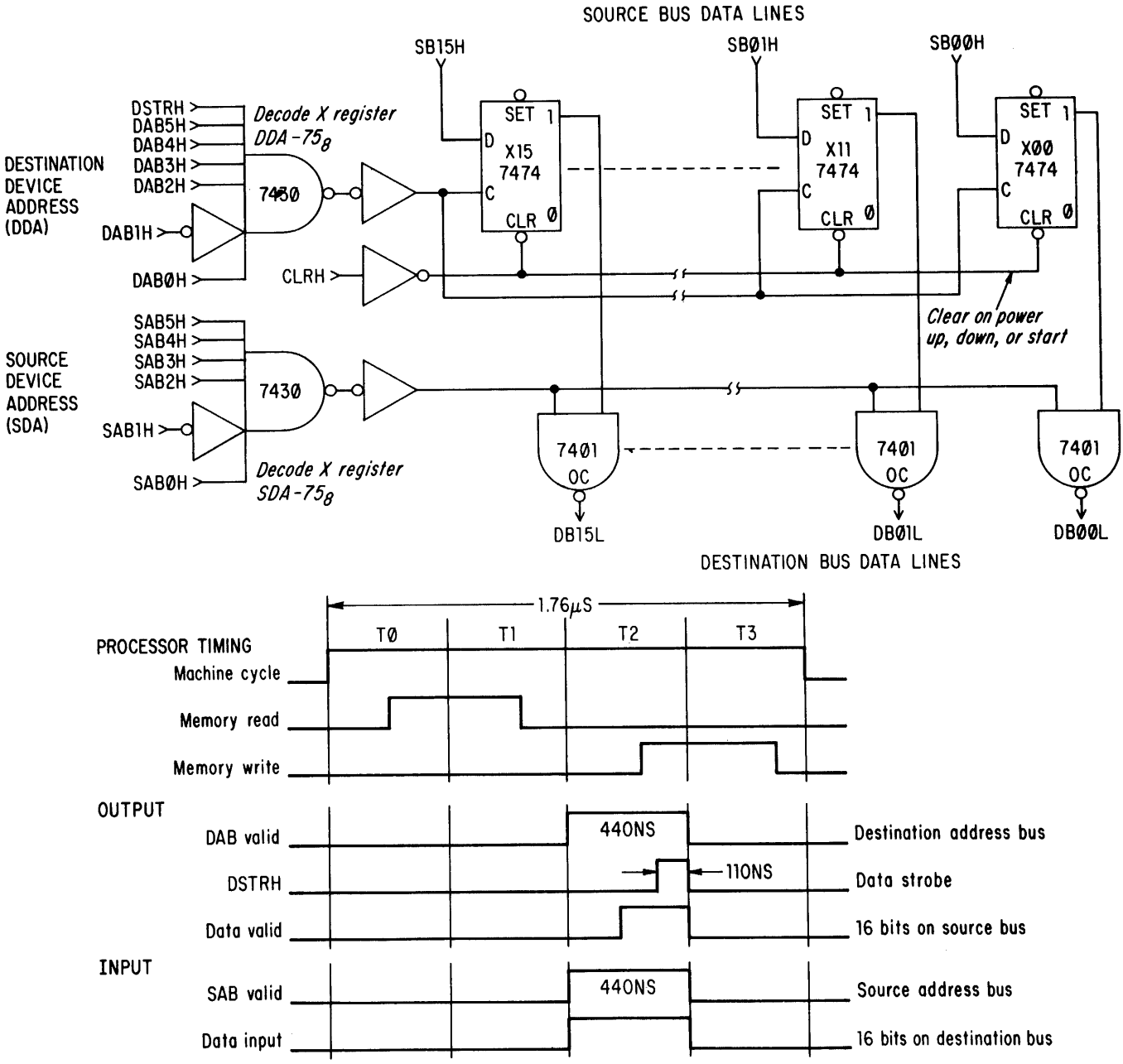


Figure 5. Programmed Data Transfers

to SAB5H. The decoding technique in a 7430 multiple input gate is identical to that for the destination address. The timing of the transfer is the same as that for output. The SAB address is valid for the same 440 ns period that a valid address appears on the DAB lines. The 110 ns strobe at the input to the receiving register occurs at the end of this interval, allowing a settling time of 330 ns to transmit the data from the DB lines through the bus modifier onto the SB lines. Whatever logic is added between the DAB decoder and the open collector gates that connect to the DB lines must not exceed two pair delays (four gates). It is recommended that any inverters used be of the high speed type such as the Signetics 8H90 (7 ns delay hex inverter) to minimize pair delays.

The DDAH and SDAH signals produced by decoding the addresses may also be used by the logic for function output or function test instructions, rather than separately decoding the same addresses for use in testing, setting or clearing flags, or controlling an IO device.

Function Generation

An FO instruction delivers up to four coded or individually usable pulses to operators for setting, clearing, or complementing flags. These pulses are placed on the control bus lines CB0H to CB3H during time 2 of the FO instruction. These lines are strobed by the combination of FUNCH, a signal present during any SF or FO instruction, DDAH, the decoded destination address, and P2H, a strobe pulse occurring at the end of time 2.

The examples in Figure 6 show three uses for the CB signals. At A, direct clear and set signals are being used to control a flipflop or flag in the device. As with the data register, CLRH may be used for a master clear when starting and during power on and power off. The type of connection at B is for clearing, setting and complementing a flag and is used with the JK type flipflop. This

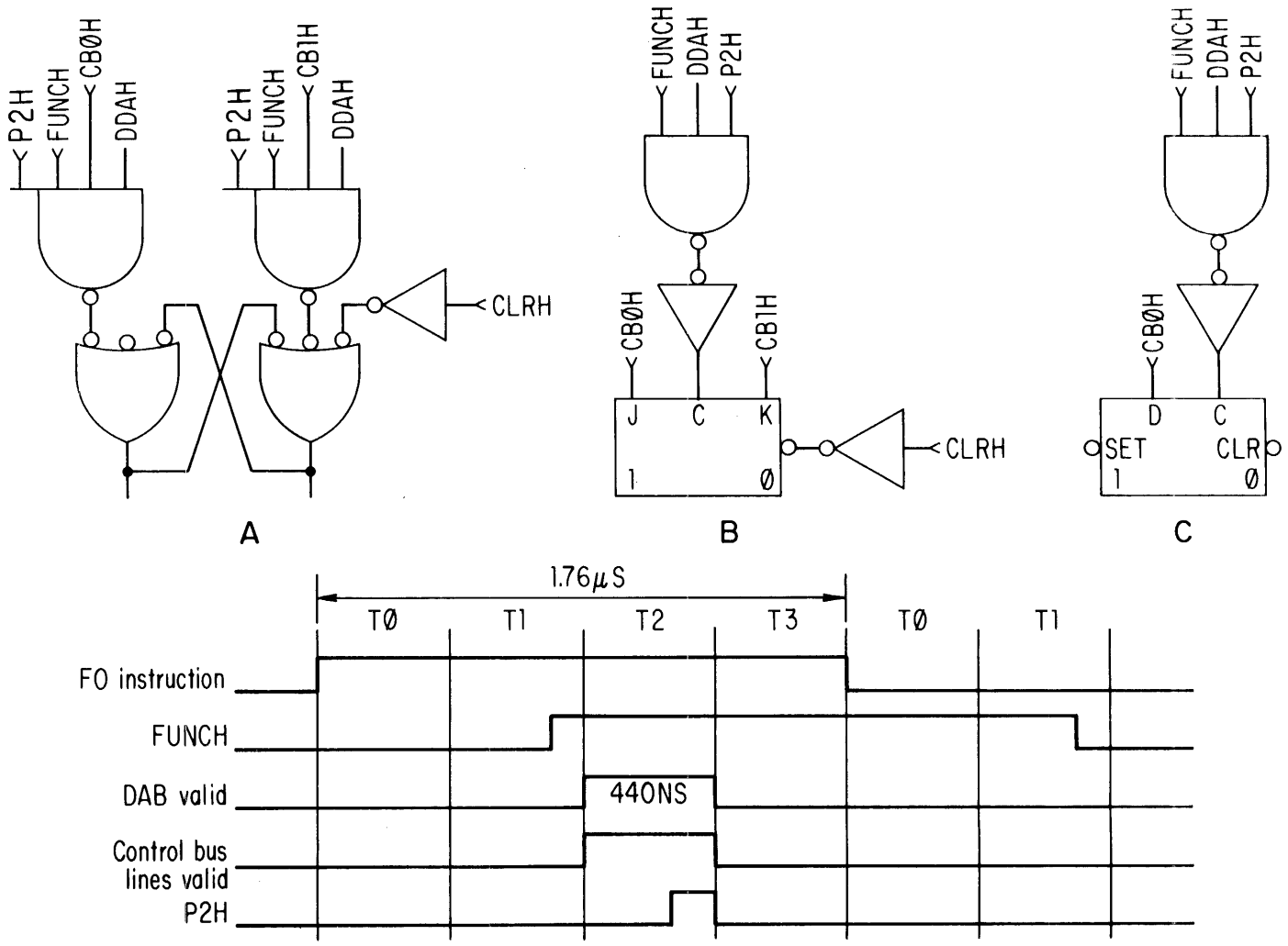


Figure 6. Function Generation

permits use of control bits 0 and 1 together to provide a microprogrammed complement of the flipflop. Example C uses a D type flipflop (7474) where the data input is connected to the CB0H line and the clock is provided by the gating of FUNCH and P2H with DDAH. This arrangement permits the transfer of the current state of the CB line, whether it be 0 or 1, into the flipflop, and it can be used to transfer up to four coded bits of data into a small function register for multiplexing or selecting up to sixteen functions.

By convention, if an operator must be placed in operation by an FO instruction, CB0H is used for this purpose. This applies to both firmware and device operators, but a simple output operator may be placed in operation just by sending data to it. Moreover the Ready flag in a device should be cleared by CB1H; if a single operator code addresses two devices, one for input, the other for output (data source and destination, respectively), Input Ready should be cleared by CB3H, Output Ready by CB1H.

Function Testing

Almost all operators contain flags, relay contacts or status levels that must be sensed by the program. These conditions are connected to the function test bus lines, FTB1L, FTB2L and FTB3L, through open collector gates type 7401 or equivalent (Figure 7). The gating function is the combination of SDAH and FUNCH. The latter is the same signal that is high during an FO instruction, but in this case

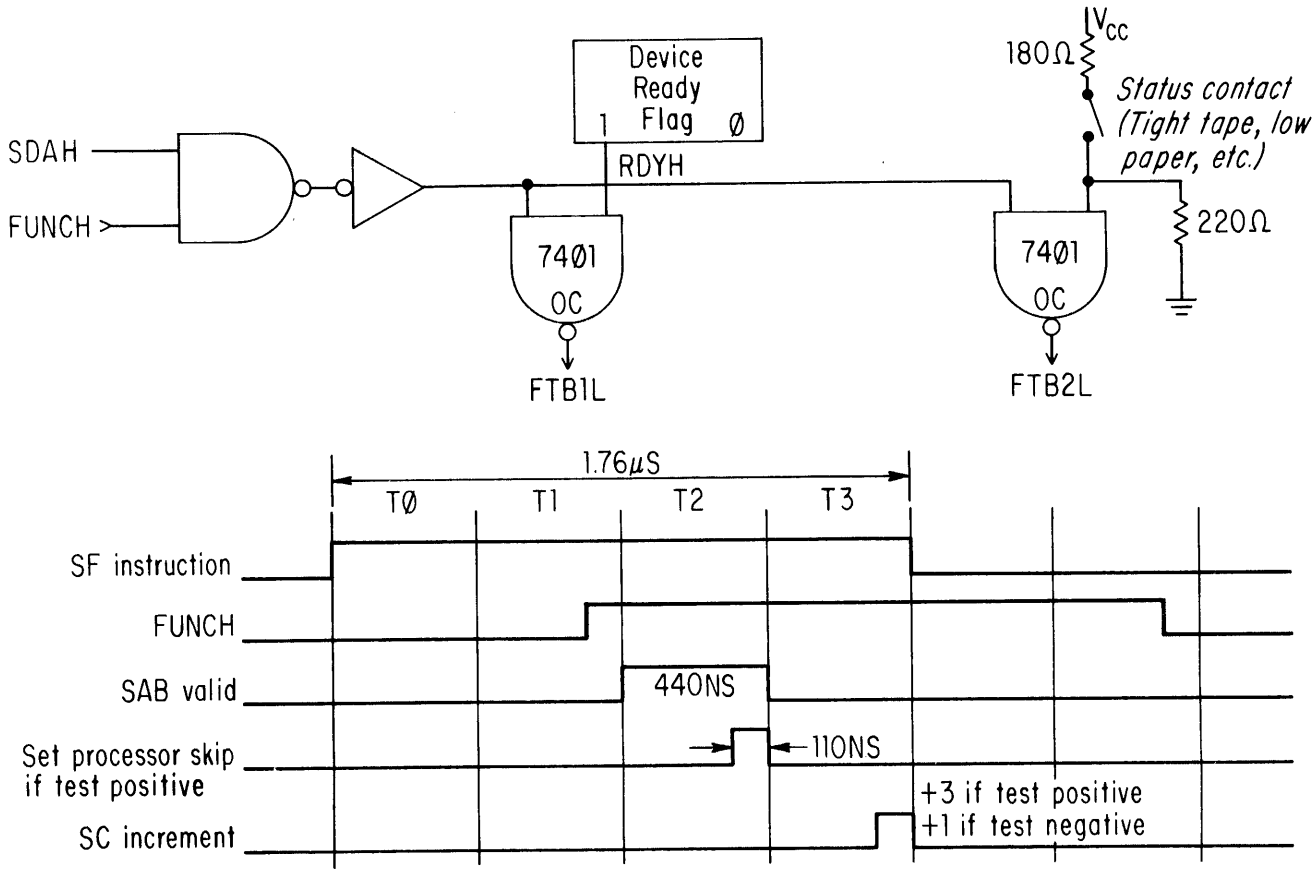


Figure 7. Function Testing

it is combined with the decoding of the appropriate source address.

At the end of the interval defined by SDAH the processor strobes the function test lines to compare the information on them with the test specification given in the SF instruction. If the test result is positive, the processor increments SC by 3 in the final time interval of the cycle; otherwise it increments SC only by 1.

By convention the Ready flag in a device is connected to FTB1L; if a single operator code addresses two devices, one for input, the other for output (data source and destination, respectively), Input Ready should be connected to FTB3L, Output Ready to FTB1L. The other lines may be assigned at the user's discretion to test conditions such as tight tape, low paper, power on, etc.

Interrupt

Any but the simplest device operator contains a flag — usually Ready — which is set at the completion of an operation to cause an interrupt. If the operator is for output, the flag is also set by CLRH so that following power on or use of the start key, the operator will indicate that it is ready to output data. In an input operator CLRH clears the flag. In either case the flag must also have provision for a programmed clear by an FO instruction.

Once RDY is set, INT REQ will set at the next ISYNH time providing the INT STAT bit for the device is on (Figure 8). INT STAT is a single bit in the interrupt status register ISR, which can be addressed as a data source or destination. The address of this register is decoded in the processor and is sent on the IO bus as IDAH when used as a destination, ISAH when used as a source. Other registers of this type may be added to the system, but decoding for them must then be done from the DAB and SAB lines.

Once INT REQ is set the interrupt bus line INTBL is pulled to ground causing the next available cycle to be used for a break. A serial signal PINL and POUTL determines which of the operators requesting an interrupt has the highest priority. Among those operators whose INT REQ flags are set, the one that is physically closest to the processor on the bus has priority.

The priority determining signal is passed along the bus from one device operator to another. If an operator receives PINL and its own INT REQ flag is clear, it generates POUTL, which is PINL at the input to the next operator. But INT REQ being clear disrupts the serial signal. Hence the first operator on the bus whose INT REQ flag is set is the only one that both receives PINL and has its own INT REQ flag set, and this is precisely the condition that allows the acknowledgement signal BKH from the processor to select an operator for a break.

BKH is gated at two different times by the external address request signal EASH, which causes a fixed-wired set of open collector gates to produce a hardwired address on the destination bus data lines during those periods of the break state when an external address is required by the processor. If no address is generated at the time of the EASH signal, the processor traps to location 0, *ie* it stores SC in location 0 and resumes operation at location 1. Otherwise the processor uses the address generated by the interrupting operator for storing SC and uses the next consecutive location to resume the execution of instructions.

Gating the address into the processor twice allows the operator to supply a different address the second time. Thus an operator might always store SC in the same pre-assigned location, but then begin program operation at various locations depending upon the cause of the interrupt.

Direct Memory Access

This feature is available to any operator in the system for passing data directly to memory, taking data directly from mem-

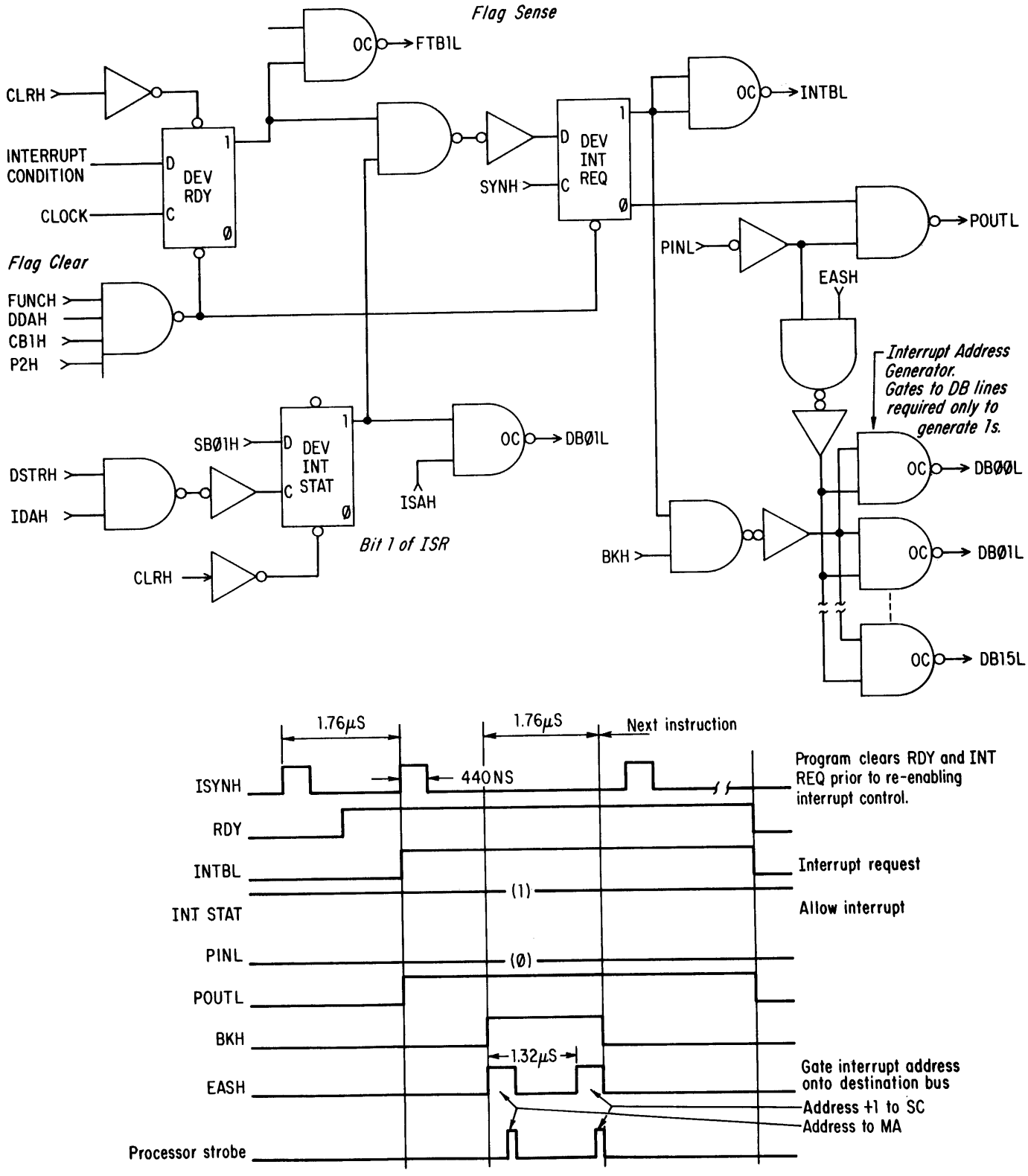


Figure 8. Interrupt

ory, or incrementing the contents of a memory location by one. Each access requires one memory cycle during which the program simply pauses. The operator requesting access must always supply the memory address.

The logic for direct memory access (Figure 9) is very similar to the logic for an interrupt except that there is no status flag. An internal condition sets DMA SYNC which in turn sets DMA REQ the next time the processor generates the ISYNH pulse, which occurs in every cycle. Setting DMA REQ gives rise to the DMA request signal DMRL on the bus. Like the interrupt logic, the DMA logic in an operator also has hardware for a serial priority determining signal that goes from one device to the next. In this case a device that receives DINL generates DOUTL for the next device if its own DMA REQ flag is clear. The setting of DMA REQ disrupts the serial signal so that it terminates at and gives priority to the first operator that both receives DINL and in which DMA REQ is set.

At the next available cycle after a request is made the processor generates the direct memory address signal EASH, which clears DMA SYNC (so that the next ISYNH clears DMA REQ) and sets DMA SERV in the device that has priority. DMA SERV combined with EASH gates a memory address onto the destination bus data lines; and combined with appropriate control levels in the device it may generate DIRBL to specify input (otherwise output is specified) or IMBL to specify that the word in the addressed memory location will be incremented.

The next operations depend upon the type of cycle. For output the processor generates EDDH to place the data from memory on the source bus data lines and sends a strobe DSTRH to load the data into a register at the operator. If the increment function is called for the processor sends the incremented word out to the operator and also writes it back in memory in place of the original data. For input the processor generates EDSH to place data from the operator on the destination bus data lines and generates a strobe internally to load the data into MB. When access is complete CLIBH clears DMA SERV to end the operation.

The logic diagram shows the basic request logic required for any type of cycle but shows the transfer logic only for input and shows only the gates for supplying a memory address. Output transfer logic is not shown and any operator making a series of DMA transfers would have a memory address counter for addressing consecutive locations.

External Instructions

This mode of operation allows a system operator to take control of the bus system and temporarily suspend the stored program. Then the operator can supply instructions from its own read-only memory in place of those from core. The instructions cannot reference memory and are executed at twice the normal rate, *ie* two per processor cycle.

Figure 10 shows the logic and timing for external instructions. The sequence begins when an FO instruction sets a flag to generate an external instruction request on the bus from the operator. An external instruction sequence follows immediately, making the FO and the sequence appear as one long instruction. The EI state is maintained until a done condition in the operator combined with EDSH drops the request.

While in the EI state, the operator must supply an instruction to the destination bus data lines every even time period. Gating is supplied by EDSH and the word is sent to IR through the bus modifier. In each odd time period the instruction is performed. Except for memory references, programs running in the EI state can accomplish the same tasks as a core program but at twice the speed.

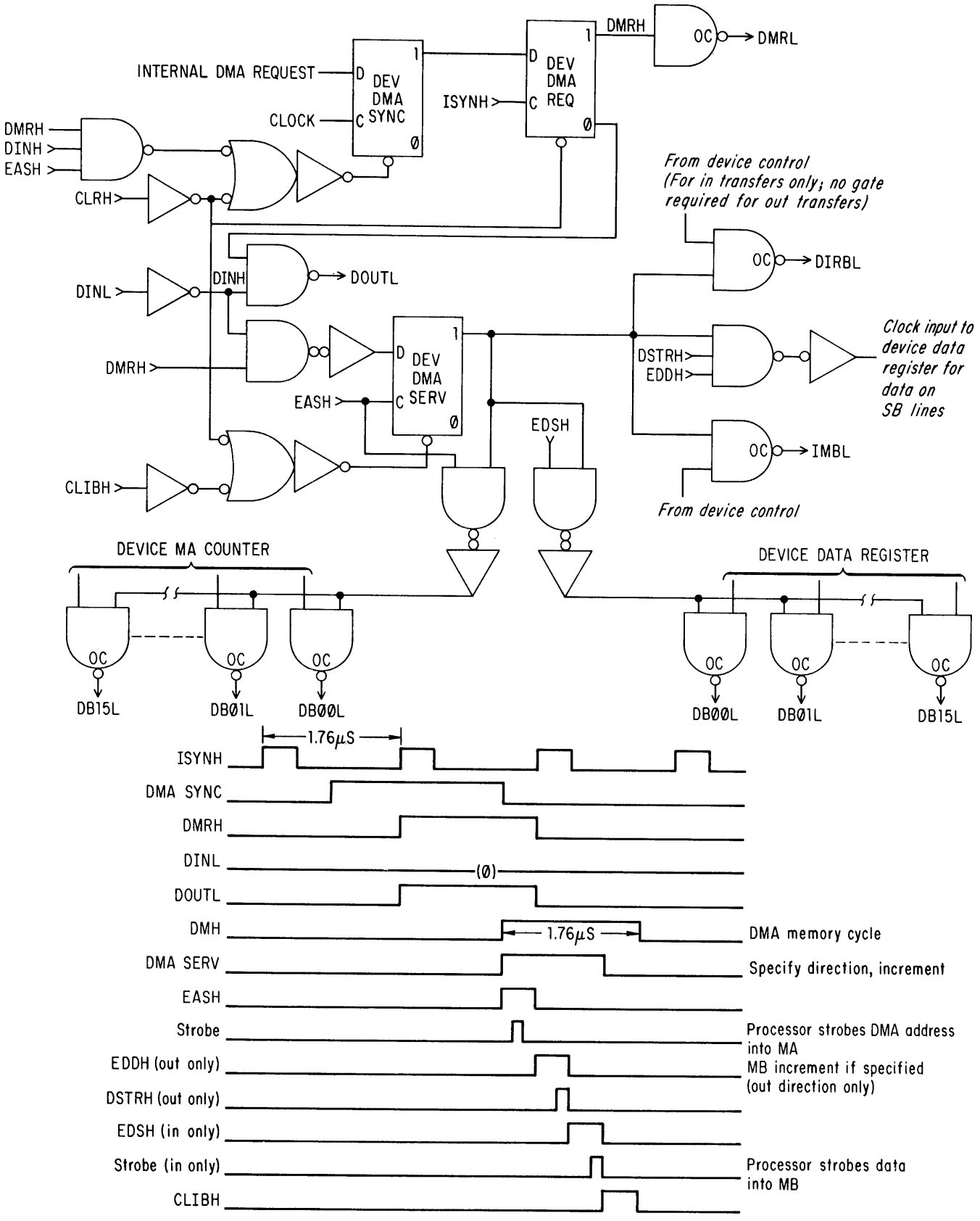


Figure 9. Direct Memory Access

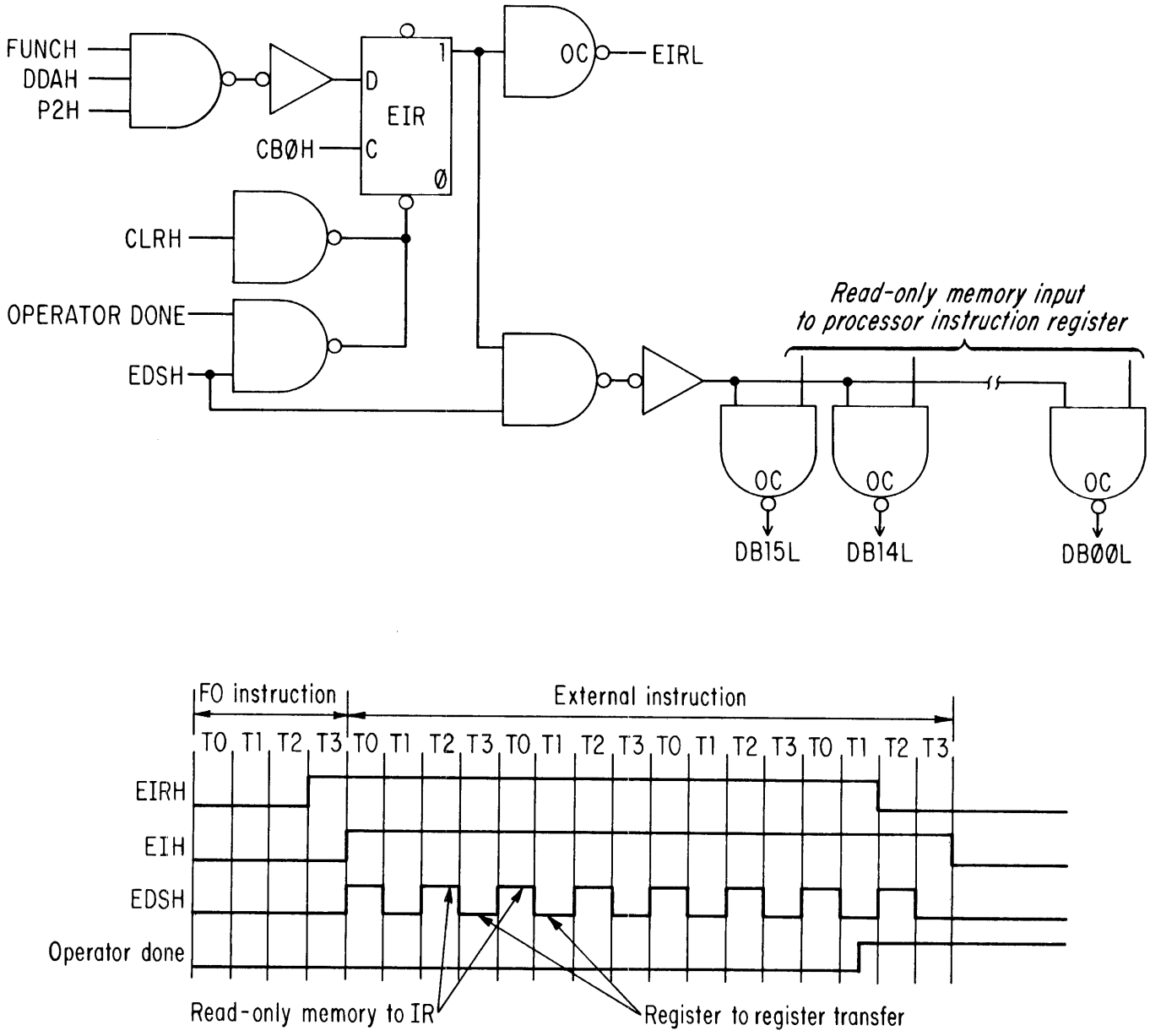


Figure 10. External Instruction Request

V DESIGN EXAMPLES

A common and relatively simple operation that is required in many process control applications is mechanical positioning of a device upon command from the computer. Figure 11 shows a valve positioner interfaced to the GRI-909 source bus. Valve position in digital form is loaded into the operator data buffer when that operator code is specified as a destination. The buffer output goes directly to the external device connector through the Amphenol plug to a D-A converter and then by cable to a positioning servo external to the computer.

Figure 12 shows an operator that monitors the status of 32 relay contacts. An FO instruction selects one of the two sets of 16 contacts, which can then be addressed as a data source, where the 16 data bits equal the relay contact status (*eg* 0 indicates an open relay, 1 indicates a closed relay).

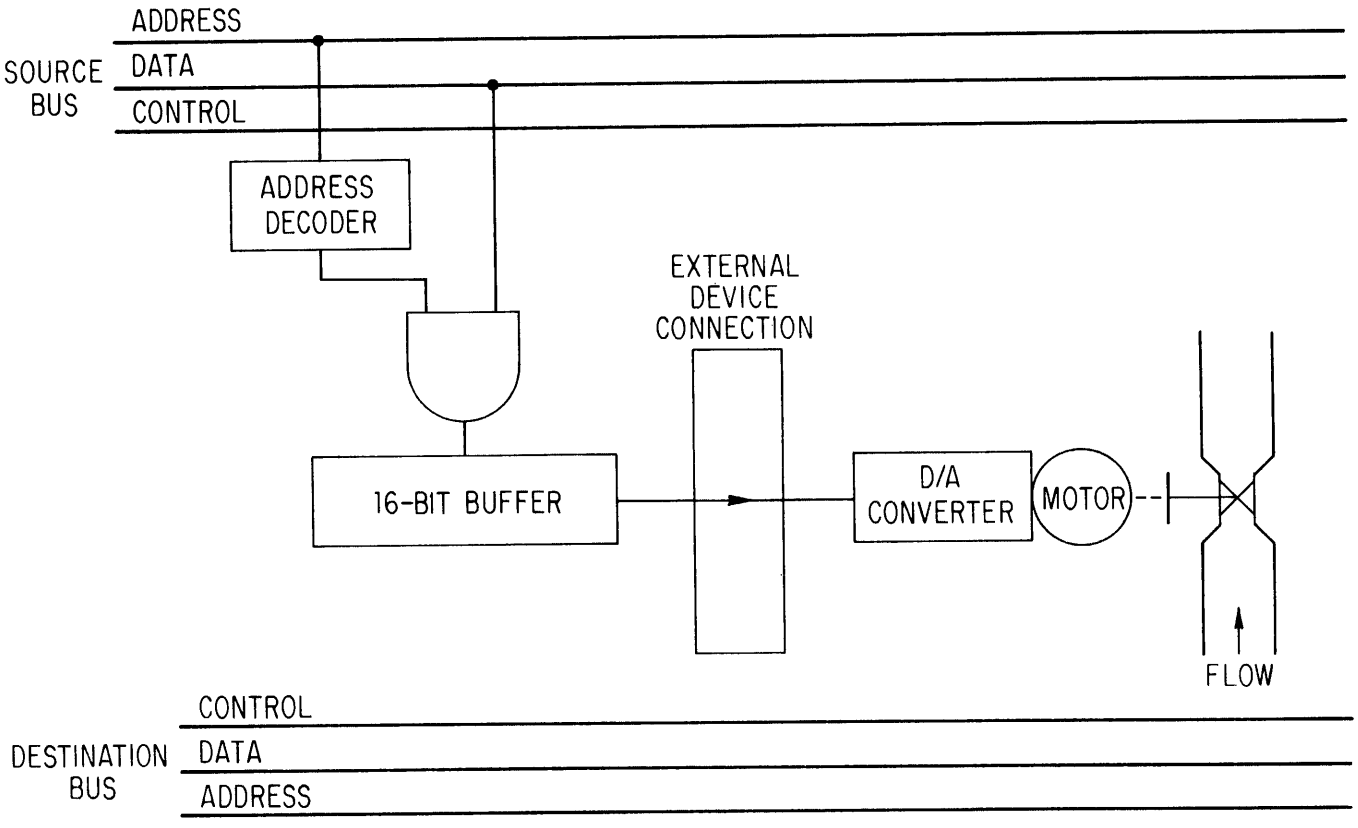


Figure 11. Valve Controller

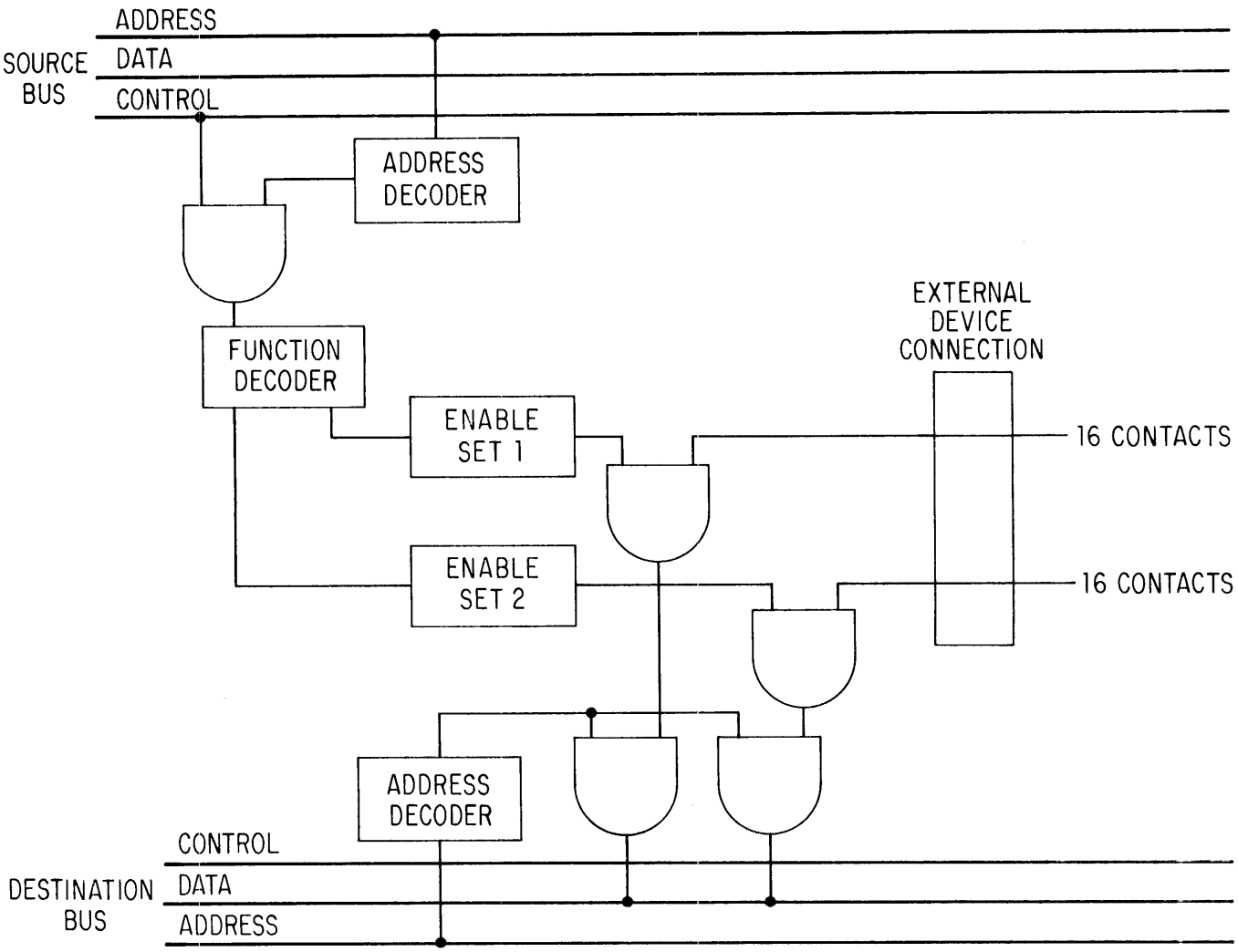


Figure 12. Relay Contact Monitor

APPENDIX C

INSTALLATION

The physical layout and dimensions of the computer chassis are shown on the next page. The chassis is set up for easy mounting in a standard 19-inch rack, either bolted in from the front or mounted on slides. (Slides can be obtained from Grant Pulley and Hardware Co., High St., Nyack, New York; catalog number SS-300-NT-20, 23½ inch travel with adjustable rear brackets.) An expansion chassis with the same dimensions can be mounted above or below the unit.

Access to the front of the chassis can be gained by swinging the hinged console aside. Viewed from the front, the power supply is on the left, the memories on the right; in the center are slots for seven large printed circuit boards. Accessible from the back of the chassis are sixteen slots for smaller boards. All cards are mounted vertically to take maximum advantage of convective cooling inside the rack. There are two types of expansion chassis: one is identical to the basic chassis, the other has connectors at the back for 24K of additional memory with its power supply.

It is recommended that the ambient temperature at the installation be maintained between 20° and 30°C, but the ambient temperature in the vicinity of the chassis can vary from 0° to 50°C without adverse effect. The relative humidity can be as high as 90%. (Although all exposed surfaces are treated to prevent corrosion, exposure of extreme humidity for long periods of time should be avoided.)

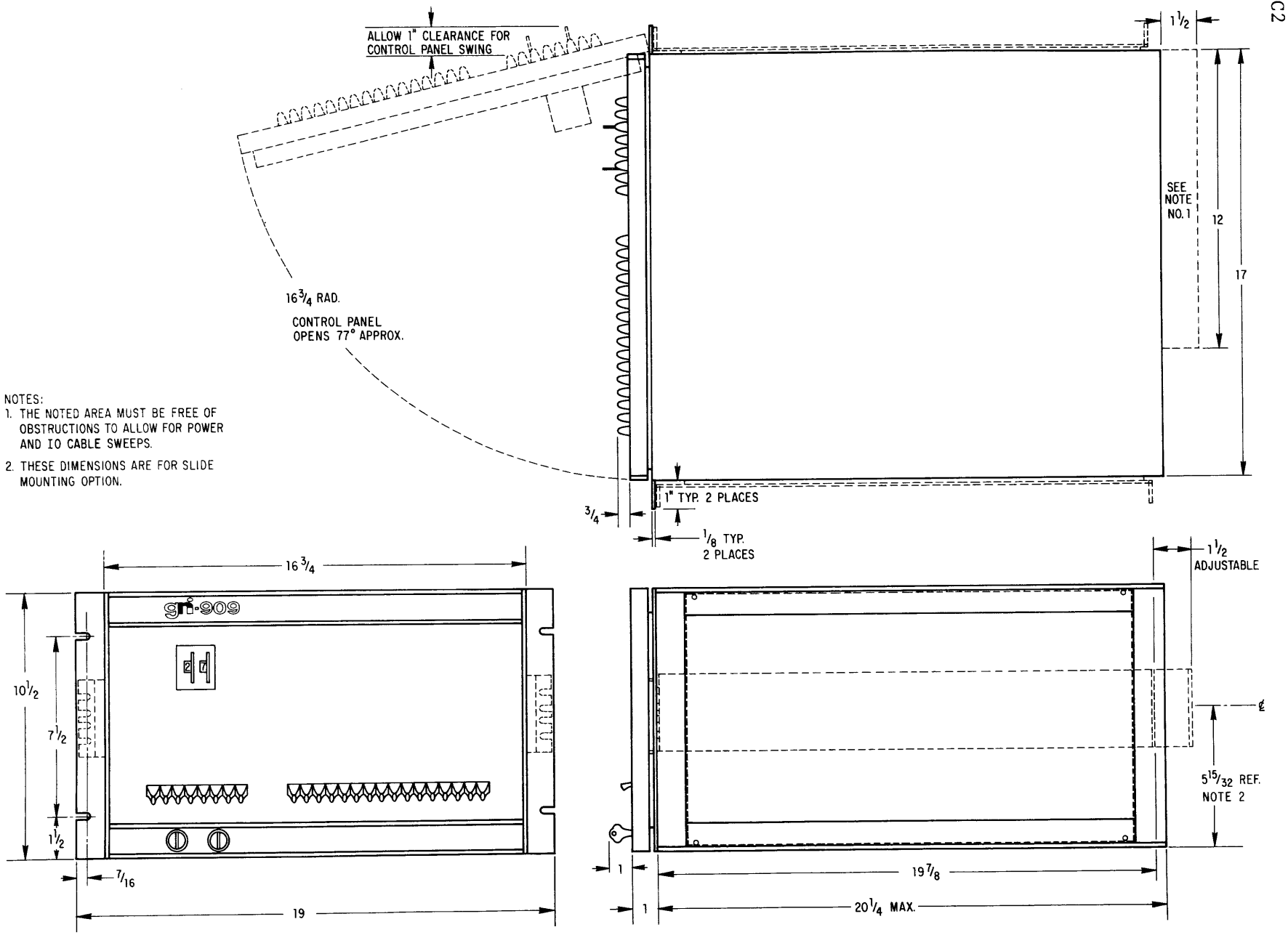
The computer uses single phase line power, 100-130 vac, 60±3% Hz. An optional power supply operates from 200-240 vac, 50±3% Hz. The power source should be capable of supplying 15 amperes. The power cable has a standard 3-wire plug and should be plugged into a properly grounded receptacle rated at 15 amperes.

	Processor	*Teletype*
Line current (115 vac)	4 amperes	2 amperes Turnon surge 7 amperes
Dissipation	150-250 watts	92 watts

The +5 vdc output of the power supply can deliver 13 amperes, of which about 6 are used by the processor with 4K of memory; the rest is available for additional memory and other optional operators. Each expansion chassis has its own power supplies.

Complete assembly instructions for the teletype are given in Section 574-100-201 of Bulletin 273B, Volume 1, *Technical Manual, 32 and 33 Teletypewriter Sets*. In particular, Part 6 of that section describes the installation of the power pack, which is mounted inside the stand as shown in the illustration on page 14. Plug the power pack cable into the pack, and plug the teletype power cable into the line power source. Plug and lock the signal cable onto the edge fingers on the back edges of the Teletype interface cards at the back of the processor chassis (the teletype output and input interface cards are at the left end, closest to the memory, and are marked TTO and TTI respectively).

	Height (inches)	*Width (inches)*	*Depth (inches)*	*Weight (pounds)*
Main chassis	10½	19	20	50
Teletype ASR33	45	22	19	56



NOTES:
1. THE NOTED AREA MUST BE FREE OF OBSTRUCTIONS TO ALLOW FOR POWER AND IO CABLE SWEEPS.
2. THESE DIMENSIONS ARE FOR SLIDE MOUNTING OPTION.

Physical Dimensions

APPENDIX D

POWER FAILURE AND AUTOMATIC RESTART

Power supply specifications are given in the preceding appendix. A rise or fall of .25 volt in the +5 volt or -5 volt output will result in a power failure. Should this occur, the power supply control circuitry signals the processor of the condition and then waits at least 100 μs before initiating an orderly power down sequence. To protect system components from being damaged by an excessive overvoltage, the +5 and -20 vdc supplies contain silicon-controlled rectifiers (known as crowbars) that will short circuit the supply if necessary.

To ensure full core memory protection, the power supply control executes an orderly power up sequence following power turnon or a power failure. If the failure condition still exists as power is coming up, the control simply initiates the shutdown sequence again. The power up sequence takes 3 seconds, and upon its completion the processor is ready to run. If the autorestart switch is on, the processor goes into normal operation starting at location 6; otherwise it comes on stopped. The switch is located on the front panel of the power supply behind the console.

Caution

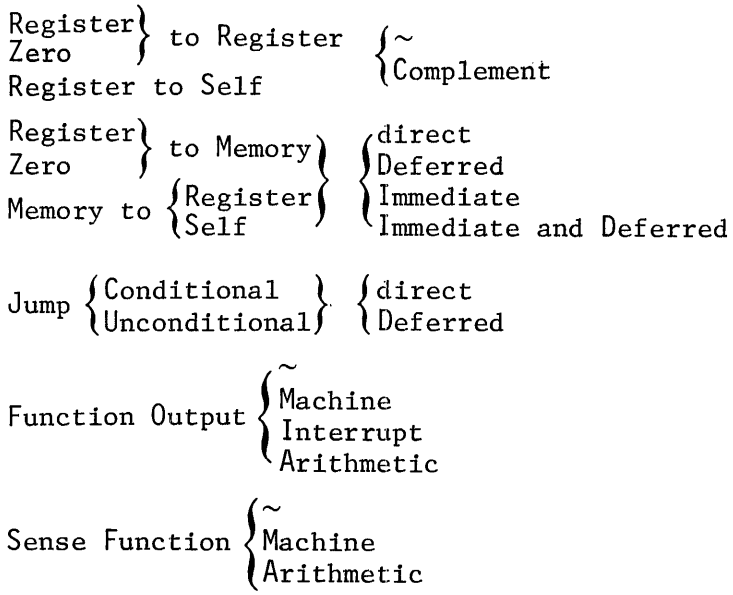
If the machine continually shuts down and restarts, the most likely cause is transients in the ac input.

APPENDIX E

INSTRUCTION MNEMONICS

Listed below are the basic instruction types with their execution times and the number of words they require. At the bottom of the page is a chart showing the derivation of the instruction mnemonics. The table on the next page lists the basic assembly mnemonics in alphabetical order. On pages E3 and E4 is a list of operator codes in numerical order with page references. Following that is a table showing the many variations of which each instruction type is capable.

	Words	*Cycles*	*Time Microseconds*
Data Transmission			
Nonmemory reference	1	1	1.76
Memory reference			
Direct	2	3	5.28
Deferred	2	4	7.04
Immediate	2	2	3.52
Immediate and deferred	2	3	5.28
Data Testing			
No jump	2	1	1.76
Jump direct	2	2	3.52
Jump deferred	2	3	5.28
Function Generating	1	1	1.76
Function Testing	1	1	1.76



INSTRUCTION MNEMONICS

ADD		0000	
AND		0100	
AO	13		
AOV		0010	
AX	11		
AY	12		
BOV		0010	
CLIF		1000	
CLL		0001	
CLOF		0011	
CML		0011	
ETZ		0100	
FO	02	0000	00
FOA	02	0000	13
FOI	02	0000	04
FOM	02	0000	00
GEZ		1010	
GR1	26		
GR2	27		
GTZ		1110	
HLT		0100	
HSP	76		
HSR	76		
ICF		0001	
ICO		0010	
IRDY		1000	
ISR	04		
JC	00	0000	03
JCD	00	0001	03
JU	00	0010	03
JUD	00	0011	03
L1		1000	
LEZ		1100	
LNK		0100	
LTZ		1000	
MPO	14		
MR	06	0000	00
MRD	06	0001	00
MRI	06	0010	00
MRID	06	0011	00
MS	06	0000	06
MSD	06	0001	06
MSI	06	0010	06
MSID	06	0011	06
MSR	17		
NEZ		0110	
NOP	00	0000	00
NOT		0001	
NPFL		1000	
OR		1100	
ORDY		0010	
P1		0100	
R1		1100	
RM	00	0000	06
RMD	00	0001	06
RMI	00	0010	06
RMID	00	0011	06
RR	00	0000	00
RRC	00	0010	00
RS	00	0000	00
RSC	00	0010	00
SC	07		
SF	00	0000	02
SFA	13	0000	02
SFM	00	0000	02
STRT		0001	
STL		0010	
SWR	10		
TRP	03		
TTI	77		
TTO	77		
XOR		1000	
ZM	00	0000	06
ZMD	00	0001	06
ZMI	00	0010	06
ZMID	00	0011	06
ZR	00	0000	00
ZRC	00	0010	00

OPERATOR CODES

Octal	Mnemonic	Page	Operator	
			Source	Destination
00			Null-Control	
01			Instruction Register	Null
02			Function Generator	Function Tester
03	TRP	2-7	Trap Register	Data Tester (*nonmemory source*) Trap Register (*memory source*)
04	ISR	2-16	Interrupt Status Register	
05			Memory Address	Null
06			Memory (Buffer)	
07	SC	2-11	Sequence Counter	
10	SWR	2-19	Console Switch Register	Null
11	AX	3-1	Register AX	
12	AY	3-1	Register AY	
13	AO	3-1	Arithmetic Operator	
14	MPO*	3-4	Multiply Operator	
15			External Data	
16			External Address	Null
17	MSR	2-11	Machine Status Register	
20				
21	BX*		Register BX	
22	BY*		Register BY	
23	BAO*		Second Arithmetic Operator	
24	BSW*	3-4	Byte Swapper	
25	BPK*	3-4	Byte Packer	
26	GR1	3-3	General Register 1	
27	GR2	3-3	General Register 2	
30				
31				
32				
33				
34				
35				
36				
37				

Octal	Mnemonic	Page	Source	Operator	Destination
40					
41					
42					
43					
44					
45					
46					
47					
50					
51					
52					
53					
54					
55					
56					
57					
60					
61					
62					
63					
64					
65					
66					
67					
70					
71					
72					
73					
74					
75	RTC*			Real Time Clock	
76	HSR HSP	4-6	High Speed Reader		High Speed Punch
77	TTI TTO	4-1	Teletype Input		Teletype Output

* Not defined in the assembler symbol table.

INSTRUCTION VARIATIONS

In order to show the wide variety of individual instructions available in the GRI-909, we list here all the variations of the basic instruction types for a single pair of operators. The assemblers recognize mnemonic or structural forms for eleven instruction types as follows:

1. Register to Register
2. Zero to Register
3. Register to Self
4. Register to Memory
5. Zero to Memory
6. Memory to Register
7. Memory to Self
8. Conditional Jump
9. Unconditional Jump
10. Function Generate
11. Sense Function

1. Register to Register

Data transmission from one register to another is the fastest and most commonly used class of machine instruction. The modifications performed on the data by the bus modifier while the data is in transit expand this form into the set listed below. For illustration the arithmetic register AX is being sent to general purpose register 1 (GR1)

Data modification	*BASE*		*FAST*
None	RR	AX,GR1	AX TO GR1
Increment by 1	RR	AX,P1,GR1	AX P1 TO GR1
Shift left 1	RR	AX,L1,GR1	AX L1 TO GR1
Shift right 1	RR	AX,R1,GR1	AX R1 TO GR1
Complement	RRC	AX,GR1	C AX TO GR1
Complement and increment by 1	RRC	AX,P1,GR1	C AX P1 TO GR1
Complement and shift left 1	RRC	AX,L1,GR1	C AX L1 TO GR1
Complement and shift right 1	RRC	AX,R1,GR1	C AX R1 TO GR1

2. Zero to Register

This is a special case of the register to register instruction where the source is the null operator. With no source register delivering data to the destination bus, the data supplied to the bus modifier is zero, and the following set of instructions results.

Data modification	*BASE*		*FAST*
None	ZR	AX	ZERO TO AX
Increment by 1	ZR	P1,AX	ZERO P1 TO AX
Shift left 1	ZR	L1,AX	ZERO L1 TO AX
Shift right 1	ZR	R1,AX	ZERO R1 TO AX
Complement	ZRC	AX	C ZERO TO AX
Complement and increment by 1	ZRC	P1,AX	C ZERO P1 TO AX
Complement and shift left 1	ZRC	L1,AX	C ZERO L1 TO AX
Complement and shift right 1	ZRC	R1,AX	C ZERO R1 TO AX

3. Register to Self

The bus modifier can be made to act directly on the contents of a single register by specifying that register as both source and destination. This special case of register to register gives this instruction set.

Data modification	*BASE*		*FAST*
Increment by 1	RS	AX,P1	AX P1
Shift left 1	RS	AX,L1	AX L1
Shift right 1	RS	AX,R1	AX R1
Complement	RSC	AX	C AX
Complement and increment by 1	RSC	AX,P1	C AX P1
Complement and shift left 1	RSC	AX,L1	C AX L1
Complement and shift right 1	RSC	AX,R1	C AX R1

4. Register to Memory

To transfer data from a register to memory the memory buffer is designated as the destination. The next consecutive memory location in the program is then used as a memory address or as an immediate data storage location. This instruction type expands into the set given here when memory addressing modes and bus modifier options are considered. AX is the source, MDATA is the address of a location containing data, and ADATA is the address of a location containing a deferred address. In immediate mode the second instruction location will be used to receive data, so we specify its contents initially as 0.

Memory addressing	*Data modification*	*BASE*		*FAST*
Direct	None	RM	AX,MDATA	AX TO MDATA
Direct	Increment by 1	RM	AX,P1,MDATA	AX P1 TO MDATA
Direct	Left 1	RM	AX,L1,MDATA	AX L1 TO MDATA
Direct	Right 1	RM	AX,R1,MDATA	AX R1 TO MDATA
Deferred	None	RMD	AX,ADATA	AX TO D ADATA
Deferred	Increment by 1	RMD	AX,P1,ADATA	AX P1 TO D ADATA
Deferred	Left 1	RMD	AX,L1ADATA	AX L1 TO D ADATA
Deferred	Right 1	RMD	AX,R1,ADATA	AX R1 TO D ADATA
Immediate	None	RMI	AX,0	AX TO I 0
Immediate	Increment by 1	RMI	AX,P1,0	AX P1 TO I 0
Immediate	Left 1	RMI	AX,L1,0	AX L1 TO I 0
Immediate	Right 1	RMI	AX,R1,0	AX R1 TO I 0
Immediate deferred	None	RMID	AX,ADATA	AX TO ID ADATA
Immediate deferred	Increment by 1	RMID	AX,P1,ADATA	AX P1 TO ID ADATA
Immediate deferred	Left 1	RMID	AX,L1,ADATA	AX L1 TO ID ADATA
Immediate deferred	Right 1	RMID	AX,R1,ADATA	AX R1 TO ID ADATA

5. Zero to Memory

This is a special case of register to memory where the source is the null operator. The source word is therefore zero and the data sent to memory is dependent upon only the operations performed in the bus modifier. The set of instructions for this special case are given using the same terminology as for register to memory.

Memory addressing	*Data modification*	*BASE*		*FAST*
Direct	None	ZM	MDATA	ZERO TO MDATA
Direct	Increment by 1	ZM	P1,MDATA	ZERO P1 TO MDATA
Direct	Left 1	ZM	L1,MDATA	ZERO L1 TO MDATA
Direct	Right 1	ZM	R1,MDATA	ZERO R1 TO MDATA
Deferred	None	ZMD	ADATA	ZERO TO D ADATA
Deferred	Increment by 1	ZMD	P1,ADATA	ZERO P1 TO D ADATA
Deferred	Left 1	ZMD	L1,ADATA	ZERO L1 TO D ADATA
Deferred	Right 1	ZMD	R1,ADATA	ZERO R1 TO D ADATA
Immediate	None	ZMI	0	ZERO TO I 0
Immediate	Increment by 1	ZMI	P1,0	ZERO P1 TO I 0

Immediate	Left 1	ZMI	L1,0	ZERO L1 TO I 0
Immediate	Right 1	ZMI	R1,0	ZERO R1 TO I 0
Immediate deferred	None	ZMID	ADATA	ZERO TO ID ADATA
Immediate deferred	Increment by 1	ZMID	P1,ADATA	ZERO P1 TO ID ADATA
Immediate deferred	Left 1	ZMID	L1,ADATA	ZERO L1 TO ID ADATA
Immediate deferred	Right 1	ZMID	R1,ADATA	ZERO R1 TO ID ADATA

6. Memory to Register

This is the exact inverse of register to memory discussed above. MDATA and ADATA have the same meaning as before, but in immediate mode we must specify the data to be supplied from the second instruction location. For an example we use 5 as the data word.

Memory addressing	*Data modification*	*BASE*		*FAST*
Direct	None	MR	MDATA,AX	MDATA TO AX
Direct	Increment by 1	MR	MDATA,P1,AX	MDATA P1 TO AX
Direct	Left 1	MR	MDATA,L1,AX	MDATA L1 TO AX
Direct	Right 1	MR	MDATA,R1,AX	MDATA R1 TO AX
Deferred	None	MRD	ADATA,AX	D ADATA TO AX
Deferred	Increment by 1	MRD	ADATA,P1,AX	D ADATA P1 TO AX
Deferred	Left 1	MRD	ADATA,L1,AX	D ADATA L1 TO AX
Deferred	Right 1	MRD	ADATA,R1,AX	D ADATA R1 TO AX
Immediate	None	MRI	5,AX	I 5 TO AX
Immediate	Increment by 1	MRI	5,P1,AX	I 5 P1 TO AX
Immediate	Left 1	MRI	5,L1,AX	I 5 L1 TO AX
Immediate	Right 1	MRI	5,R1,AX	I 5 R1 TO AX
Immediate deferred	None	MRID	ADATA,AX	ID ADATA TO AX
Immediate deferred	Increment by 1	MRID	ADATA,P1,AX	ID ADATA P1 TO AX
Immediate deferred	Left 1	MRID	ADATA,L1,AX	ID ADATA L1 TO AX
Immediate deferred	Right 1	MRID	ADATA,R1,AX	ID ADATA R1 TO AX

7. Memory to Self

A memory reference instruction can access only one memory data location, but when the memory buffer is designated as both source and destination, the contents of that location can be modified directly, giving this instruction set.

Memory addressing	*Data modification*	*BASE*		*FAST*
Direct	Increment by 1	MS	MDATA,P1	MDATA P1
Direct	Left 1	MS	MDATA,L1	MDATA L1
Direct	Right 1	MS	MDATA,R1	MDATA R1
Deferred	None	MSD	ADATA	D ADATA
Deferred	Increment by 1	MSD	ADATA,P1	D ADATA P1
Deferred	Left 1	MSD	ADATA,L1	D ADATA L1
Deferred	Right 1	MSD	ADATA,R1	D ADATA R1
Immediate	None	MSI	0	I 0
Immediate	Increment by 1	MSI	0,P1	I 0 P1
Immeiiate	Left 1	MSI	0,L1	I 0 L1
Immediate	Right 1	MSI	0,R1	I 0 R1
Immediate deferred	None	MSID	ADATA	ID ADATA
Immediate deferred	Increment by 1	MSID	ADATA,P1	ID ADATA P1
Immediate deferred	Left 1	MSID	ADATA,L1	ID ADATA L1
Immediate deferred	Right 1	MSID	ADATA,R1	ID ADATA R1

8. Conditional Jump

This is a data test instruction. The source data is send directly to the data tester and is not subject to action by the bus modifier. Jump addressing and conditions for testing the data produce this instruction set.

Memory addressing	*Data test*	*BASE*		*FAST*
Direct	Data = 0	JC	AX,ETZ,BEGIN	IF AX ETZ GO TO BEGIN
Direct	Data < 0	JC	AX,LTZ,BEGIN	IF AX LTZ GO TO BEGIN
Direct	Data ≤ 0	JC	AX,LEZ,BEGIN	IF AX LEZ GO TO BEGIN
Direct	Data ≠ 0	JC	AX,NEZ,BEGIN	IF AX NEZ GO TO BEGIN
Direct	Data ≥ 0	JC	AX,GEZ,BEGIN	IF AX GEZ GO TO BEGIN
Direct	Data > 0	JC	AX,GTZ,BEGIN	IF AX GTZ GO TO BEGIN
Deferred	Data = 0	JCD	AX,ETZ,ABGIN	IF AX ETZ GO TO D ABGIN
Deferred	Data < 0	JCD	AX,LTZ,ABGIN	IF AX LTZ GO TO D ABGIN
Deferred	Data ≤ 0	JCD	AX,LEZ,ABGIN	IF AX LEZ GO TO D ABGIN
Deferred	Data ≠ 0	JCD	AX,NEZ,ABGIN	IF AX NEZ GO TO D ABGIN
Deferred	Data ≥ 0	JCD	AX,GEZ,ABGIN	IF AX GEZ GO TO D ABGIN
Deferred	Data > 0	JCD	AX,GTZ,ABGIN	IF AX GTZ GO TO D ABGIN

9. Unconditional Jump

Memory addressing	*BASE*	*FAST*
Direct	JU BEGIN	GO TO BEGIN
Deferred	JUD ABGIN	GO TO D ABGIN

10. Function Generate

The four function bits in an FO instruction word permit up to sixteen unique FO instructions per destination operator address. These bit combinations can be given symbolic names or can simply be written in octal in the BASE language. The instruction set shown here generates functions for an A-D converter, mnemonic ADC.

BASE	*FAST*	*BASE*	*FAST*
FO 0,ADC	N0 TO ADC	FO 10,ADC	N10 TO ADC
FO 1,ADC	N1 TO ADC	FO 11,ADC	N11 TO ADC
FO 2,ADC	N2 TO ADC	FO 12,ADC	N12 TO ADC
FO 3,ADC	N3 TO ADC	FO 13,ADC	N13 TO ADC
FO 4,ADC	N4 TO ADC	FO 14,ADC	N14 TO ADC
FO 5,ADC	N5 TO ADC	FO 15,ADC	N15 TO ADC
FO 6,ADC	N6 TO ADC	FO 16,ADC	N16 TO ADC
FO 7,ADC	N7 TO ADC	FO 17,ADC	N17 TO ADC

11. Sense Function

The four control bits in an SF instruction word allow up to fourteen different skip instructions per source operator address. The rightmost bit determines whether the skip shall occur if any condition specified by the other three is satisfied, or if no condition specified by them is satisfied. The conditions may be given in octal as is done here in BASE or by a symbolic name as must be done in the FAST language.

BASE	*FAST*	*BASE*	*FAST*
SF ADC,1	SKIP IF ADC F1	SF ADC,NOT 1	SKIP IF ADC NOT F1
SF ADC,2	SKIP IF ADC F2	SF ADC,NOT 2	SKIP IF ADC NOT F2
SF ADC,3	SKIP IF ADC F3	SF ADC,NOT 3	SKIP IF ADC NOT F3
SF ADC,4	SKIP IF ADC F4	SF ADC,NOT 4	SKIP IF ADC NOT F4
SF ADC,5	SKIP IF ADC F5	SF ADC,NOT 5	SKIP IF ADC NOT F5
SF ADC,6	SKIP IF ADC F6	SF ADC,NOT 6	SKIP IF ADC NOT F6
SF ADC,7	SKIP IF ADC F7	SF ADC,NOT 7	SKIP IF ADC NOT F7

APPENDIX F

IN-OUT CODES

The table below lists the in-out devices, their interrupt status bit and channel assignments, mnemonics, operator codes and GRI option numbers. The table beginning on the next page lists the complete teletype code. Codes generated by the keyboard have a 1 in the most significant bit, but this bit can be 0 or 1 in a code sent to the printer. The lower case character set (codes 340-376) is not available on the Model 33, but giving one of these codes causes the teletype to print the corresponding upper case character. Definitions of control codes are those given by ASCII. Most control codes, however, have no effect on the teletype, and their definitions bear no necessary relation to the use of the codes in conjunction with the GRI-909 software.

IN-OUT DEVICES

Interrupt Status Bit	Trap Location	Device	Mnemonic	Operator Code	Page	Option Number
	0	Power failure		00	2-19	
	0	Breakpoint		01	2-17	
0	11	Teletype output	TTO	77	4-1	S42-002
1	14	Teletype input	TTI	77	4-1	S42-001
2	17	High speed punch	HSP	76	4-6	S42-006
3	22	High speed reader	HSR	76	4-6	S42-004
4						
5						
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						

TELETYPE CODE

8-Bit Octal Code	Character	Remarks
200	NUL	Null, tape feed. Control shift P.
201	SOH	Start of heading; also SOM, start of message. Control A.
202	STX	Start of text; also EOA, end of address. Control B.
203	ETX	End of text; also EOM, end of message. Control C.
204	EOT	End of transmission; shuts off TWX machines. Control D.
205	ENQ	Enquiry; also WRU, "Who are You?" Triggers identification ("Here is ...".) at remote station if so equipped. Control E.
206	ACK	Acknowledge; Also RU, "Are you ...?" Control F.
207	BEL	Rings the bell. Control G.
210	BS	Backspace; also FEO, format effector. Backspaces some machines. Control H.
211	HT	Horizontal tab. Control I.
212	LF	Line feed or line space; advances paper to next line. Duplicated by control J.
213	VT	Vertical tab. Control K.
214	FF	Form feed to top of next page. Control L.
215	CR	Carriage return to beginning of line. Control M.
216	SO	Shift out; changes ribbon color to red. Control N.
217	SI	Shift in; changes ribbon color to black. Control O.
220	DLE	Data link escape. Control P (DC0).
221	DC1	Device control 1, turns transmitter (reader) on. Control Q (X ON).
222	DC2	Device control 2, turns punch or auxiliary on. Control R (TAPE, AUX ON).
223	DC3	Device control 3, turns transmitter (reader) off. Control S (X OFF).
224	DC4	Device control 4, turns punch or auxiliary off. Control T (AUX OFF).
225	NAK	Negative acknowledge; also ERR, error. Control U.
226	SYN	Synchronous idle. Control V.
227	ETB	End of transmission block; also LEM, logical end of medium. Control W.
230	CAN	Cancel. Control X.
231	EM	End of medium. Control Y.
232	SUB	Substitute. Control Z.

8-Bit Octal Code	Character	Remarks
233	ESC	Escape, prefix. This code is also generated by control shift K.
234	FS	File separator. Control shift L.
235	GS	Group separator. Control shift M.
236	RS	Record separator. Control shift N.
237	US	Unit separator. Control shift O.
240	SP	Space.
241	!	
242	"	
243	#	
244	$	
245	%	
246	&	
247	'	Accent acute or apostrophe.
250	(	
251	)	
252	*	
253	+	
254	,	
255	-	
256	.	
257	/	
260	0	
261	1	
262	2	
263	3	
264	4	
265	5	
266	6	
267	7	
270	8	
271	9	
272	:	
273	;	

8-Bit Octal Code	Character	Remarks
274	<	
275	=	
276	>	
277	?	
300	@	
301	A	
302	B	
303	C	
304	D	
305	E	
306	F	
307	G	
310	H	
311	I	
312	J	
313	K	
314	L	
315	M	
316	N	
317	O	
320	P	
321	Q	
322	R	
323	S	
324	T	
325	U	
326	V	
327	W	
330	X	
331	Y	
332	Z	
333	[	Shift K.
334	\	Shift L.

8-Bit Octal Code	Character	Remarks
335	]	Shift M.
336	↑	
337	←	
340	`	Accent grave.
341	a	
342	b	
343	c	
344	d	
345	e	
346	f	
347	g	
350	h	
351	i	
352	j	
353	k	
354	l	
355	m	
356	n	
357	o	
360	p	
361	q	
362	r	
363	s	
364	t	
365	u	
366	v	
367	w	
370	x	
371	y	
372	z	
373	{	
374	|	

8-Bit Octal Code	Character	Remarks
375	}	On early versions, either of these codes may be generated by either the ALT MODE or ESC key.
376	~	
377	DEL	Delete, rub out.

Keys That Generate No Codes

REPT	Causes any other key that is struck to repeat continously until REPT is released.
LOC LF	Local line feed.
LOC CR	Local carriage return.
BREAK	Opens the line (machine sends a continuous string of null characters).
BRK RLS	Break release (not applicable).
HERE IS	Transmits predetermined 20-character message.

APPENDIX G

NUMERICAL TABLES

POWERS OF TWO IN DECIMAL

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25

POWERS OF TEN IN OCTAL

10^n	n	10^{-n}
1	0	1.000 000 000 000 000 000 00
12	1	0.063 146 314 631 463 146 31
144	2	0.005 075 341 217 270 243 66
1 750	3	0.000 406 111 564 570 651 77
23 420	4	0.000 032 155 613 530 704 15
303 240	5	0.000 002 476 132 610 706 64
3 641 100	6	0.000 000 206 157 364 055 37
46 113 200	7	0.000 000 015 327 745 152 75
575 360 400	8	0.000 000 001 257 143 561 06
7 346 545 000	9	0.000 000 000 104 560 276 41
112 402 762 000	10	0.000 000 000 006 676 337 66
1 351 035 564 000	11	0.000 000 000 000 537 657 77
16 432 451 210 000	12	0.000 000 000 000 043 136 32
221 411 634 520 000	13	0.000 000 000 000 003 411 35
2 657 142 036 440 000	14	0.000 000 000 000 000 264 11
34 327 724 461 500 000	15	0.000 000 000 000 000 022 01
434 157 115 760 200 000	16	0.000 000 000 000 000 001 63
5 432 127 413 542 400 000	17	0.000 000 000 000 000 000 14
67 405 553 164 731 000 000	18	0.000 000 000 000 000 000 01

OCTAL TO DECIMAL CONVERSION, INTEGERS

Octal 0000-0777
Decimal 0000-0511

Octal	Decimal
10000	4096
20000	8192
30000	12288
40000	16384
50000	20480
60000	24576
70000	28672

	0	1	2	3	4	5	6	7
0000	0000	0001	0002	0003	0004	0005	0006	0007
0010	0008	0009	0010	0011	0012	0013	0014	0015
0020	0016	0017	0018	0019	0020	0021	0022	0023
0030	0024	0025	0026	0027	0028	0029	0030	0031
0040	0032	0033	0034	0035	0036	0037	0038	0039
0050	0040	0041	0042	0043	0044	0045	0046	0047
0060	0048	0049	0050	0051	0052	0053	0054	0055
0070	0056	0057	0058	0059	0060	0061	0062	0063
0100	0064	0065	0066	0067	0068	0069	0070	0071
0110	0072	0073	0074	0075	0076	0077	0078	0079
0120	0080	0081	0082	0083	0084	0085	0086	0087
0130	0088	0089	0090	0091	0092	0093	0094	0095
0140	0096	0097	0098	0099	0100	0101	0102	0103
0150	0104	0105	0106	0107	0108	0109	0110	0111
0160	0112	0113	0114	0115	0116	0117	0118	0119
0170	0120	0121	0122	0123	0124	0125	0126	0127
0200	0128	0129	0130	0131	0132	0133	0134	0135
0210	0136	0137	0138	0139	0140	0141	0142	0143
0220	0144	0145	0146	0147	0148	0149	0150	0151
0230	0152	0153	0154	0155	0156	0157	0158	0159
0240	0160	0161	0162	0163	0164	0165	0166	0167
0250	0168	0169	0170	0171	0172	0173	0174	0175
0260	0176	0177	0178	0179	0180	0181	0182	0183
0270	0184	0185	0186	0187	0188	0189	0190	0191
0300	0192	0193	0194	0195	0196	0197	0198	0199
0310	0200	0201	0202	0203	0204	0205	0206	0207
0320	0208	0209	0210	0211	0212	0213	0214	0215
0330	0216	0217	0218	0219	0220	0221	0222	0223
0340	0224	0225	0226	0227	0228	0229	0230	0231
0350	0232	0233	0234	0235	0236	0237	0238	0239
0360	0240	0241	0242	0243	0244	0245	0246	0247
0370	0248	0249	0250	0251	0252	0253	0254	0255

	0	1	2	3	4	5	6	7
0400	0256	0257	0258	0259	0260	0261	0262	0263
0410	0264	0265	0266	0267	0268	0269	0270	0271
0420	0272	0273	0274	0275	0276	0277	0278	0279
0430	0280	0281	0282	0283	0284	0285	0286	0287
0440	0288	0289	0290	0291	0292	0293	0294	0295
0450	0296	0297	0298	0299	0300	0301	0302	0303
0460	0304	0305	0306	0307	0308	0309	0310	0311
0470	0312	0313	0314	0315	0316	0317	0318	0319
0500	0320	0321	0322	0323	0324	0325	0326	0327
0510	0328	0329	0330	0331	0332	0333	0334	0335
0520	0336	0337	0338	0339	0340	0341	0342	0343
0530	0344	0345	0346	0347	0348	0349	0350	0351
0540	0352	0353	0354	0355	0356	0357	0358	0359
0550	0360	0361	0362	0363	0364	0365	0366	0367
0560	0368	0369	0370	0371	0372	0373	0374	0375
0570	0376	0377	0378	0379	0380	0381	0382	0383
0600	0384	0385	0386	0387	0388	0389	0390	0391
0610	0392	0393	0394	0395	0396	0397	0398	0399
0620	0400	0401	0402	0403	0404	0405	0406	0407
0630	0408	0409	0410	0411	0412	0413	0414	0415
0640	0416	0417	0418	0419	0420	0421	0422	0423
0650	0424	0425	0426	0427	0428	0429	0430	0431
0660	0432	0433	0434	0435	0436	0437	0438	0439
0670	0440	0441	0442	0443	0444	0445	0446	0447
0700	0448	0449	0450	0451	0452	0453	0454	0455
0710	0456	0457	0458	0459	0460	0461	0462	0463
0720	0464	0465	0466	0467	0468	0469	0470	0471
0730	0472	0473	0474	0475	0476	0477	0478	0479
0740	0480	0481	0482	0483	0484	0485	0486	0487
0750	0488	0489	0490	0491	0492	0493	0494	0495
0760	0496	0497	0498	0499	0500	0501	0502	0503
0770	0504	0505	0506	0507	0508	0509	0510	0511

Octal 1000-1777
Decimal 0512-1023

	0	1	2	3	4	5	6	7
1000	0512	0513	0514	0515	0516	0517	0518	0519
1010	0520	0521	0522	0523	0524	0525	0526	0527
1020	0528	0529	0530	0531	0532	0533	0534	0535
1030	0536	0537	0538	0539	0540	0541	0542	0543
1040	0544	0545	0546	0547	0548	0549	0550	0551
1050	0552	0553	0554	0555	0556	0557	0558	0559
1060	0560	0561	0562	0563	0564	0565	0566	0567
1070	0568	0569	0570	0571	0572	0573	0574	0575
1100	0576	0577	0578	0579	0580	0581	0582	0583
1110	0584	0585	0586	0587	0588	0589	0590	0591
1120	0592	0593	0594	0595	0596	0597	0598	0599
1130	0600	0601	0602	0603	0604	0605	0606	0607
1140	0608	0609	0610	0611	0612	0613	0614	0615
1150	0616	0617	0618	0619	0620	0621	0622	0623
1160	0624	0625	0626	0627	0628	0629	0630	0631
1170	0632	0633	0634	0635	0636	0637	0638	0639
1200	0640	0641	0642	0643	0644	0645	0646	0647
1210	0648	0649	0650	0651	0652	0653	0654	0655
1220	0656	0657	0658	0659	0660	0661	0662	0663
1230	0664	0665	0666	0667	0668	0669	0670	0671
1240	0672	0673	0674	0675	0676	0677	0678	0679
1250	0680	0681	0682	0683	0684	0685	0686	0687
1260	0688	0689	0690	0691	0692	0693	0694	0695
1270	0696	0697	0698	0699	0700	0701	0702	0703
1300	0704	0705	0706	0707	0708	0709	0710	0711
1310	0712	0713	0714	0715	0716	0717	0718	0719
1320	0720	0721	0722	0723	0724	0725	0726	0727
1330	0728	0729	0730	0731	0732	0733	0734	0735
1340	0736	0737	0738	0739	0740	0741	0742	0743
1350	0744	0745	0746	0747	0748	0749	0750	0751
1360	0752	0753	0754	0755	0756	0757	0758	0759
1370	0760	0761	0762	0763	0764	0765	0766	0767

	0	1	2	3	4	5	6	7
1400	0768	0769	0770	0771	0772	0773	0774	0775
1410	0776	0777	0778	0779	0780	0781	0782	0783
1420	0784	0785	0786	0787	0788	0789	0790	0791
1430	0792	0793	0794	0795	0796	0797	0798	0799
1440	0800	0801	0802	0803	0804	0805	0806	0807
1450	0808	0809	0810	0811	0812	0813	0814	0815
1460	0816	0817	0818	0819	0820	0821	0822	0823
1470	0824	0825	0826	0827	0828	0829	0830	0831
1500	0832	0833	0834	0835	0836	0837	0838	0839
1510	0840	0841	0842	0843	0844	0845	0846	0847
1520	0848	0849	0850	0851	0852	0853	0854	0855
1530	0856	0857	0858	0859	0860	0861	0862	0863
1540	0864	0865	0866	0867	0868	0869	0870	0871
1550	0872	0873	0874	0875	0876	0877	0878	0879
1560	0880	0881	0882	0883	0884	0885	0886	0887
1570	0888	0889	0890	0891	0892	0893	0894	0895
1600	0896	0897	0898	0899	0900	0901	0902	0903
1610	0904	0905	0906	0907	0908	0909	0910	0911
1620	0912	0913	0914	0915	0916	0917	0918	0919
1630	0920	0921	0922	0923	0924	0925	0926	0927
1640	0928	0929	0930	0931	0932	0933	0934	0935
1650	0936	0937	0938	0939	0940	0941	0942	0943
1660	0944	0945	0946	0947	0948	0949	0950	0951
1670	0952	0953	0954	0955	0956	0957	0958	0959
1700	0960	0961	0962	0963	0964	0965	0966	0967
1710	0968	0969	0970	0971	0972	0973	0974	0975
1720	0976	0977	0978	0979	0980	0981	0982	0983
1730	0984	0985	0986	0987	0988	0989	0990	0991
1740	0992	0993	0994	0995	0996	0997	0998	0999
1750	1000	1001	1002	1003	1004	1005	1006	1007
1760	1008	1009	1010	1011	1012	1013	1014	1015
1770	1016	1017	1018	1019	1020	1021	1022	1023

Octal to Decimal Conversion, Integers

	0	1	2	3	4	5	6	7
2000	1024	1025	1026	1027	1028	1029	1030	1031
2010	1032	1033	1034	1035	1036	1037	1038	1039
2020	1040	1041	1042	1043	1044	1045	1046	1047
2030	1048	1049	1050	1051	1052	1053	1054	1055
2040	1056	1057	1058	1059	1060	1061	1062	1063
2050	1064	1065	1066	1067	1068	1069	1070	1071
2060	1072	1073	1074	1075	1076	1077	1078	1079
2070	1080	1081	1082	1083	1084	1085	1086	1087
2100	1088	1089	1090	1091	1092	1093	1094	1095
2110	1096	1097	1098	1099	1100	1101	1102	1103
2120	1104	1105	1106	1107	1108	1109	1110	1111
2130	1112	1113	1114	1115	1116	1117	1118	1119
2140	1120	1121	1122	1123	1124	1125	1126	1127
2150	1128	1129	1130	1131	1132	1133	1134	1135
2160	1136	1137	1138	1139	1140	1141	1142	1143
2170	1144	1145	1146	1147	1148	1149	1150	1151
2200	1152	1153	1154	1155	1156	1157	1158	1159
2210	1160	1161	1162	1163	1164	1165	1166	1167
2220	1168	1169	1170	1171	1172	1173	1174	1175
2230	1176	1177	1178	1179	1180	1181	1182	1183
2240	1184	1185	1186	1187	1188	1189	1190	1191
2250	1192	1193	1194	1195	1196	1197	1198	1199
2260	1200	1201	1202	1203	1204	1205	1206	1207
2270	1208	1209	1210	1211	1212	1213	1214	1215
2300	1216	1217	1218	1219	1220	1221	1222	1223
2310	1224	1225	1226	1227	1228	1229	1230	1231
2320	1232	1233	1234	1235	1236	1237	1238	1239
2330	1240	1241	1242	1243	1244	1245	1246	1247
2340	1248	1249	1250	1251	1252	1253	1254	1255
2350	1256	1257	1258	1259	1260	1261	1262	1263
2360	1264	1265	1266	1267	1268	1269	1270	1271
2370	1272	1273	1274	1275	1276	1277	1278	1279

	0	1	2	3	4	5	6	7
2400	1280	1281	1282	1283	1284	1285	1286	1287
2410	1288	1289	1290	1291	1292	1293	1294	1295
2420	1296	1297	1298	1299	1300	1301	1302	1303
2430	1304	1305	1306	1307	1308	1309	1310	1311
2440	1312	1313	1314	1315	1316	1317	1318	1319
2450	1320	1321	1322	1323	1324	1325	1326	1327
2460	1328	1329	1330	1331	1332	1333	1334	1335
2470	1336	1337	1338	1339	1340	1341	1342	1343
2500	1344	1345	1346	1347	1348	1349	1350	1351
2510	1352	1353	1354	1355	1356	1357	1358	1359
2520	1360	1361	1362	1363	1364	1365	1366	1367
2530	1368	1369	1370	1371	1372	1373	1374	1375
2540	1376	1377	1378	1379	1380	1381	1382	1383
2550	1384	1385	1386	1387	1388	1389	1390	1391
2560	1392	1393	1394	1395	1396	1397	1398	1399
2570	1400	1401	1402	1403	1404	1405	1406	1407
2600	1408	1409	1410	1411	1412	1413	1414	1415
2610	1416	1417	1418	1419	1420	1421	1422	1423
2620	1424	1425	1426	1427	1428	1429	1430	1431
2630	1432	1433	1434	1435	1436	1437	1438	1439
2640	1440	1441	1442	1443	1444	1445	1446	1447
2650	1448	1449	1450	1451	1452	1453	1454	1455
2660	1456	1457	1458	1459	1460	1461	1462	1463
2670	1464	1465	1466	1467	1468	1469	1470	1471
2700	1472	1473	1474	1475	1476	1477	1478	1479
2710	1480	1481	1482	1483	1484	1485	1486	1487
2720	1488	1489	1490	1491	1492	1493	1494	1495
2730	1496	1497	1498	1499	1500	1501	1502	1503
2740	1504	1505	1506	1507	1508	1509	1510	1511
2750	1512	1513	1514	1515	1516	1517	1518	1519
2760	1520	1521	1522	1523	1524	1525	1526	1527
2770	1528	1529	1530	1531	1532	1533	1534	1535

Octal 2000-2777
Decimal 1024-1535

Octal	*Decimal*
10000	4096
20000	8192
30000	12288
40000	16384
50000	20480
60000	24576
70000	28672

	0	1	2	3	4	5	6	7
3000	1536	1537	1538	1539	1540	1541	1542	1543
3010	1544	1545	1546	1547	1548	1549	1550	1551
3020	1552	1553	1554	1555	1556	1557	1558	1559
3030	1560	1561	1562	1563	1564	1565	1566	1567
3040	1568	1569	1570	1571	1572	1573	1574	1575
3050	1576	1577	1578	1579	1580	1581	1582	1583
3060	1584	1585	1586	1587	1588	1589	1590	1591
3070	1592	1593	1594	1595	1596	1597	1598	1599
3100	1600	1601	1602	1603	1604	1605	1606	1607
3110	1608	1609	1610	1611	1612	1613	1614	1615
3120	1616	1617	1618	1619	1620	1621	1622	1623
3130	1624	1625	1626	1627	1628	1629	1630	1631
3140	1632	1633	1634	1635	1636	1637	1638	1639
3150	1640	1641	1642	1643	1644	1645	1646	1647
3160	1648	1649	1650	1651	1652	1653	1654	1655
3170	1656	1657	1658	1659	1660	1661	1662	1663
3200	1664	1665	1666	1667	1668	1669	1670	1671
3210	1672	1673	1674	1675	1676	1677	1678	1679
3220	1680	1681	1682	1683	1684	1685	1686	1687
3230	1688	1689	1690	1691	1692	1693	1694	1695
3240	1696	1697	1698	1699	1700	1701	1702	1703
3250	1704	1705	1706	1707	1708	1709	1710	1711
3260	1712	1713	1714	1715	1716	1717	1718	1719
3270	1720	1721	1722	1723	1724	1725	1726	1727
3300	1728	1729	1730	1731	1732	1733	1734	1735
3310	1736	1737	1738	1739	1740	1741	1742	1743
3320	1744	1745	1746	1747	1748	1749	1750	1751
3330	1752	1753	1754	1755	1756	1757	1758	1759
3340	1760	1761	1762	1763	1764	1765	1766	1767
3350	1768	1769	1770	1771	1772	1773	1774	1775
3360	1776	1777	1778	1779	1780	1781	1782	1783
3370	1784	1785	1786	1787	1788	1789	1790	1791

	0	1	2	3	4	5	6	7
3400	1792	1793	1794	1795	1796	1797	1798	1799
3410	1800	1801	1802	1803	1804	1805	1806	1807
3420	1808	1809	1810	1811	1812	1813	1814	1815
3430	1816	1817	1818	1819	1820	1821	1822	1823
3440	1824	1825	1826	1827	1828	1829	1830	1831
3450	1832	1833	1834	1835	1836	1837	1838	1839
3460	1840	1841	1842	1843	1844	1845	1846	1847
3470	1848	1849	1850	1851	1852	1853	1854	1855
3500	1856	1857	1858	1859	1860	1861	1862	1863
3510	1864	1865	1866	1867	1868	1869	1870	1871
3520	1872	1873	1874	1875	1876	1877	1878	1879
3530	1880	1881	1882	1883	1884	1885	1886	1887
3540	1888	1889	1890	1891	1892	1893	1894	1895
3550	1896	1897	1898	1899	1900	1901	1902	1903
3560	1904	1905	1906	1907	1908	1909	1910	1911
3570	1912	1913	1914	1915	1916	1917	1918	1919
3600	1920	1921	1922	1923	1924	1925	1926	1927
3610	1928	1929	1930	1931	1932	1933	1934	1935
3620	1936	1937	1938	1939	1940	1941	1942	1943
3630	1944	1945	1946	1947	1948	1949	1950	1951
3640	1952	1953	1954	1955	1956	1957	1958	1959
3650	1960	1961	1962	1963	1964	1965	1966	1967
3660	1968	1969	1970	1971	1972	1973	1974	1975
3670	1976	1977	1978	1979	1980	1981	1982	1983
3700	1984	1985	1986	1987	1988	1989	1990	1991
3710	1992	1993	1994	1995	1996	1997	1998	1999
3720	2000	2001	2002	2003	2004	2005	2006	2007
3730	2008	2009	2010	2011	2012	2013	2014	2015
3740	2016	2017	2018	2019	2020	2021	2022	2023
3750	2024	2025	2026	2027	2028	2029	2030	2031
3760	2032	2033	2034	2035	2036	2037	2038	2039
3770	2040	2041	2042	2043	2044	2045	2046	2047

Octal 3000-3777
Decimal 1536-2047

Octal to Decimal Conversion, Integers

Octal 4000-4777
Decimal 2048-2559

Octal	*Decimal*
10000	4096
20000	8192
30000	12288
40000	16384
50000	20480
60000	24576
70000	28672

	0	1	2	3	4	5	6	7
4000	2048	2049	2050	2051	2052	2053	2054	2055
4010	2056	2057	2058	2059	2060	2061	2062	2063
4020	2064	2065	2066	2067	2068	2069	2070	2071
4030	2072	2073	2074	2075	2076	2077	2078	2079
4040	2080	2081	2082	2083	2084	2085	2086	2087
4050	2088	2089	2090	2091	2092	2093	2094	2095
4060	2096	2097	2098	2099	2100	2101	2102	2103
4070	2104	2105	2106	2107	2108	2109	2110	2111
4100	2112	2113	2114	2115	2116	2117	2118	2119
4110	2120	2121	2122	2123	2124	2125	2126	2127
4120	2128	2129	2130	2131	2132	2133	2134	2135
4130	2136	2137	2138	2139	2140	2141	2142	2143
4140	2144	2145	2146	2147	2148	2149	2150	2151
4150	2152	2153	2154	2155	2156	2157	2158	2159
4160	2160	2161	2162	2163	2164	2165	2166	2167
4170	2168	2169	2170	2171	2172	2173	2174	2175
4200	2176	2177	2178	2179	2180	2181	2182	2183
4210	2184	2185	2186	2187	2188	2189	2190	2191
4220	2192	2193	2194	2195	2196	2197	2198	2199
4230	2200	2201	2202	2203	2204	2205	2206	2207
4240	2208	2209	2210	2211	2212	2213	2214	2215
4250	2216	2217	2218	2219	2220	2221	2222	2223
4260	2224	2225	2226	2227	2228	2229	2230	2231
4270	2232	2233	2234	2235	2236	2237	2238	2239
4300	2240	2241	2242	2243	2244	2245	2246	2247
4310	2248	2249	2250	2251	2252	2253	2254	2255
4320	2256	2257	2258	2259	2260	2261	2262	2263
4330	2264	2265	2266	2267	2268	2269	2270	2271
4340	2272	2273	2274	2275	2276	2277	2278	2279
4350	2280	2281	2282	2283	2284	2285	2286	2287
4360	2288	2289	2290	2291	2292	2293	2294	2295
4370	2296	2297	2298	2299	2300	2301	2302	2303

	0	1	2	3	4	5	6	7
4400	2304	2305	2306	2307	2308	2309	2310	2311
4410	2312	2313	2314	2315	2316	2317	2318	2319
4420	2320	2321	2322	2323	2324	2325	2326	2327
4430	2328	2329	2330	2331	2332	2333	2334	2335
4440	2336	2337	2338	2339	2340	2341	2342	2343
4450	2344	2345	2346	2347	2348	2349	2350	2351
4460	2352	2353	2354	2355	2356	2357	2358	2359
4470	2360	2361	2362	2363	2364	2365	2366	2367
4500	2368	2369	2370	2371	2372	2373	2374	2375
4510	2376	2377	2378	2379	2380	2381	2382	2383
4520	2384	2385	2386	2387	2388	2389	2390	2391
4530	2392	2393	2394	2395	2396	2397	2398	2399
4540	2400	2401	2402	2403	2404	2405	2406	2407
4550	2408	2409	2410	2411	2412	2413	2414	2415
4560	2416	2417	2418	2419	2420	2421	2422	2423
4570	2424	2425	2426	2427	2428	2429	2430	2431
4600	2432	2433	2434	2435	2436	2437	2438	2439
4610	2440	2441	2442	2443	2444	2445	2446	2447
4620	2448	2449	2450	2451	2452	2453	2454	2455
4630	2456	2457	2458	2459	2460	2461	2462	2463
4640	2464	2465	2466	2467	2468	2469	2470	2471
4650	2472	2473	2474	2475	2476	2477	2478	2479
4660	2480	2481	2482	2483	2484	2485	2486	2487
4670	2488	2489	2490	2491	2492	2493	2494	2495
4700	2496	2497	2498	2499	2500	2501	2502	2503
4710	2504	2505	2506	2507	2508	2509	2510	2511
4720	2512	2513	2514	2515	2516	2517	2518	2519
4730	2520	2521	2522	2523	2524	2525	2526	2527
4740	2528	2529	2530	2531	2532	2533	2534	2535
4750	2536	2537	2538	2539	2540	2541	2542	2543
4760	2544	2545	2546	2547	2548	2549	2550	2551
4770	2552	2553	2554	2555	2556	2557	2558	2559

Octal 5000-5777
Decimal 2560-3071

	0	1	2	3	4	5	6	7
5000	2560	2561	2562	2563	2564	2565	2566	2567
5010	2568	2569	2570	2571	2572	2573	2574	2575
5020	2576	2577	2578	2579	2580	2581	2582	2583
5030	2584	2585	2586	2587	2588	2589	2590	2591
5040	2592	2593	2594	2595	2596	2597	2598	2599
5050	2600	2601	2602	2603	2604	2605	2606	2607
5060	2608	2609	2610	2611	2612	2613	2614	2615
5070	2616	2617	2618	2619	2620	2621	2622	2623
5100	2624	2625	2626	2627	2628	2629	2630	2631
5110	2632	2633	2634	2635	2636	2637	2638	2639
5120	2640	2641	2642	2643	2644	2645	2646	2647
5130	2648	2649	2650	2651	2652	2653	2654	2655
5140	2656	2657	2658	2659	2660	2661	2662	2663
5150	2664	2665	2666	2667	2668	2669	2670	2671
5160	2672	2673	2674	2675	2676	2677	2678	2679
5170	2680	2681	2682	2683	2684	2685	2686	2687
5200	2688	2689	2690	2691	2692	2693	2694	2695
5210	2696	2697	2698	2699	2700	2701	2702	2703
5220	2704	2705	2706	2707	2708	2709	2710	2711
5230	2712	2713	2714	2715	2716	2717	2718	2719
5240	2720	2721	2722	2723	2724	2725	2726	2727
5250	2728	2729	2730	2731	2732	2733	2734	2735
5260	2736	2737	2738	2739	2740	2741	2742	2743
5270	2744	2745	2746	2747	2748	2749	2750	2751
5300	2752	2753	2754	2755	2756	2757	2758	2759
5310	2760	2761	2762	2763	2764	2765	2766	2767
5320	2768	2769	2770	2771	2772	2773	2774	2775
5330	2776	2777	2778	2779	2780	2781	2782	2783
5340	2784	2785	2786	2787	2788	2789	2790	2791
5350	2792	2793	2794	2795	2796	2797	2798	2799
5360	2800	2801	2802	2803	2804	2805	2806	2807
5370	2808	2809	2810	2811	2812	2813	2814	2815

	0	1	2	3	4	5	6	7
5400	2816	2817	2818	2819	2820	2821	2822	2823
5410	2824	2825	2826	2827	2828	2829	2830	2831
5420	2832	2833	2834	2835	2836	2837	2838	2839
5430	2840	2841	2842	2843	2844	2845	2846	2847
5440	2848	2849	2850	2851	2852	2853	2854	2855
5450	2856	2857	2858	2859	2860	2861	2862	2863
5460	2864	2865	2866	2867	2868	2869	2870	2871
5470	2872	2873	2874	2875	2876	2877	2878	2879
5500	2880	2881	2882	2883	2884	2885	2886	2887
5510	2888	2889	2890	2891	2892	2893	2894	2895
5520	2896	2897	2898	2899	2900	2901	2902	2903
5530	2904	2905	2906	2907	2908	2909	2910	2911
5540	2912	2913	2914	2915	2916	2917	2918	2919
5550	2920	2921	2922	2923	2924	2925	2926	2927
5560	2928	2929	2930	2931	2932	2933	2934	2935
5570	2936	2937	2938	2939	2940	2941	2942	2943
5600	2944	2945	2946	2947	2948	2949	2950	2951
5610	2952	2953	2954	2955	2956	2957	2958	2959
5620	2960	2961	2962	2963	2964	2965	2966	2967
5630	2968	2969	2970	2971	2972	2973	2974	2975
5640	2976	2977	2978	2979	2980	2981	2982	2983
5650	2984	2985	2986	2987	2988	2989	2990	2991
5660	2992	2993	2994	2995	2996	2997	2998	2999
5670	3000	3001	3002	3003	3004	3005	3006	3007
5700	3008	3009	3010	3011	3012	3013	3014	3015
5710	3016	3017	3018	3019	3020	3021	3022	3023
5720	3024	3025	3026	3027	3028	3029	3030	3031
5730	3032	3033	3034	3035	3036	3037	3038	3039
5740	3040	3041	3042	3043	3044	3045	3046	3047
5750	3048	3049	3050	3051	3052	3053	3054	3055
5760	3056	3057	3058	3059	3060	3061	3062	3063
5770	3064	3065	3066	3067	3068	3069	3070	3071

Octal to Decimal Conversion, Integers

	0	1	2	3	4	5	6	7
6000	3072	3073	3074	3075	3076	3077	3078	3079
6010	3080	3081	3082	3083	3084	3085	3086	3087
6020	3088	3089	3090	3091	3092	3093	3094	3095
6030	3096	3097	3098	3099	3100	3101	3102	3103
6040	3104	3105	3106	3107	3108	3109	3110	3111
6050	3112	3113	3114	3115	3116	3117	3118	3119
6060	3120	3121	3122	3123	3124	3125	3126	3127
6070	3128	3129	3130	3131	3132	3133	3134	3135
6100	3136	3137	3138	3139	3140	3141	3142	3143
6110	3144	3145	3146	3147	3148	3149	3150	3151
6120	3152	3153	3154	3155	3156	3157	3158	3159
6130	3160	3161	3162	3163	3164	3165	3166	3167
6140	3168	3169	3170	3171	3172	3173	3174	3175
6150	3176	3177	3178	3179	3180	3181	3182	3183
6160	3184	3185	3186	3187	3188	3189	3190	3191
6170	3192	3193	3194	3195	3196	3197	3198	3199
6200	3200	3201	3202	3203	3204	3205	3206	3207
6210	3208	3209	3210	3211	3212	3213	3214	3215
6220	3216	3217	3218	3219	3220	3221	3222	3223
6230	3224	3225	3226	3227	3228	3229	3230	3231
6240	3232	3233	3234	3235	3236	3237	3238	3239
6250	3240	3241	3242	3243	3244	3245	3246	3247
6260	3248	3249	3250	3251	3252	3253	3254	3255
6270	3256	3257	3258	3259	3260	3261	3262	3263
6300	3264	3265	3266	3267	3268	3269	3270	3271
6310	3272	3273	3274	3275	3276	3277	3278	3279
6320	3280	3281	3282	3283	3284	3285	3286	3287
6330	3288	3289	3290	3291	3292	3293	3294	3295
6340	3296	3297	3298	3299	3300	3301	3302	3303
6350	3304	3305	3306	3307	3308	3309	3310	3311
6360	3312	3313	3314	3315	3316	3317	3318	3319
6370	3320	3321	3322	3323	3324	3325	3326	3327

	0	1	2	3	4	5	6	7
6400	3328	3329	3330	3331	3332	3333	3334	3335
6410	3336	3337	3338	3339	3340	3341	3342	3343
6420	3344	3345	3346	3347	3348	3349	3350	3351
6430	3352	3353	3354	3355	3356	3357	3358	3359
6440	3360	3361	3362	3363	3364	3365	3366	3367
6450	3368	3369	3370	3371	3372	3373	3374	3375
6460	3376	3377	3378	3379	3380	3381	3382	3383
6470	3384	3385	3386	3387	3388	3389	3390	3391
6500	3392	3393	3394	3395	3396	3397	3398	3399
6510	3400	3401	3402	3403	3404	3405	3406	3407
6520	3408	3409	3410	3411	3412	3413	3414	3415
6530	3416	3417	3418	3419	3420	3421	3422	3423
6540	3424	3425	3426	3427	3428	3429	3430	3431
6550	3432	3433	3434	3435	3436	3437	3438	3439
6560	3440	3441	3442	3443	3444	3445	3446	3447
6570	3448	3449	3450	3451	3452	3453	3454	3455
6600	3456	3457	3458	3459	3460	3461	3462	3463
6610	3464	3465	3466	3467	3468	3469	3470	3471
6620	3472	3473	3474	3475	3476	3477	3478	3479
6630	3480	3481	3482	3483	3484	3485	3486	3487
6640	3488	3489	3490	3491	3492	3493	3494	3495
6650	3496	3497	3498	3499	3500	3501	3502	3503
6660	3504	3505	3506	3507	3508	3509	3510	3511
6670	3512	3513	3514	3515	3516	3517	3518	3519
6700	3520	3521	3522	3523	3524	3525	3526	3527
6710	3528	3529	3530	3531	3532	3533	3534	3535
6720	3536	3537	3538	3539	3540	3541	3542	3543
6730	3544	3545	3546	3547	3548	3549	3550	3551
6740	3552	3553	3554	3555	3556	3557	3558	3559
6750	3560	3561	3562	3563	3564	3565	3566	3567
6760	3568	3569	3570	3571	3572	3573	3574	3575
6770	3576	3577	3578	3579	3580	3581	3582	3583

Octal 6000-6777
Decimal 3072-3583

Octal	*Decimal*
10000	4096
20000	8192
30000	12288
40000	16384
50000	20480
60000	24576
70000	28672

	0	1	2	3	4	5	6	7
7000	3584	3585	3586	3587	3588	3589	3590	3591
7010	3592	3593	3594	3595	3596	3597	3598	3599
7020	3600	3601	3602	3603	3604	3605	3606	3607
7030	3608	3609	3610	3611	3612	3613	3614	3615
7040	3616	3617	3618	3619	3620	3621	3622	3623
7050	3624	3625	3626	3627	3628	3629	3630	3631
7060	3632	3633	3634	3635	3636	3637	3638	3639
7070	3640	3641	3642	3643	3644	3645	3646	3647
7100	3648	3649	3650	3651	3652	3653	3654	3655
7110	3656	3657	3658	3659	3660	3661	3662	3663
7120	3664	3665	3666	3667	3668	3669	3670	3671
7130	3672	3673	3674	3675	3676	3677	3678	3679
7140	3680	3681	3682	3683	3684	3685	3686	3687
7150	3688	3689	3690	3691	3692	3693	3694	3695
7160	3696	3697	3698	3699	3700	3701	3702	3703
7170	3704	3705	3706	3707	3708	3709	3710	3711
7200	3712	3713	3714	3715	3716	3717	3718	3719
7210	3720	3721	3722	3723	3724	3725	3726	3727
7220	3728	3729	3730	3731	3732	3733	3734	3735
7230	3736	3737	3738	3739	3740	3741	3742	3743
7240	3744	3745	3746	3747	3748	3749	3750	3751
7250	3752	3753	3754	3755	3756	3757	3758	3759
7260	3760	3761	3762	3763	3764	3765	3766	3767
7270	3768	3769	3770	3771	3772	3773	3774	3775
7300	3776	3777	3778	3779	3780	3781	3782	3783
7310	3784	3785	3786	3787	3788	3789	3790	3791
7320	3792	3793	3794	3795	3796	3797	3798	3799
7330	3800	3801	3802	3803	3804	3805	3806	3807
7340	3808	3809	3810	3811	3812	3813	3814	3815
7350	3816	3817	3818	3819	3820	3821	3822	3823
7360	3824	3825	3826	3827	3828	3829	3830	3831
7370	3832	3833	3834	3835	3836	3837	3838	3839

	0	1	2	3	4	5	6	7
7400	3840	3841	3842	3843	3844	3845	3846	3847
7410	3848	3849	3850	3851	3852	3853	3854	3855
7420	3856	3857	3858	3859	3860	3861	3862	3863
7430	3864	3865	3866	3867	3868	3869	3870	3871
7440	3872	3873	3874	3875	3876	3877	3878	3879
7450	3880	3881	3882	3883	3884	3885	3886	3887
7460	3888	3889	3890	3891	3892	3893	3894	3895
7470	3896	3897	3898	3899	3900	3901	3902	3903
7500	3904	3905	3906	3907	3908	3909	3910	3911
7510	3912	3913	3914	3915	3916	3917	3918	3919
7520	3920	3921	3922	3923	3924	3925	3926	3927
7530	3928	3929	3930	3931	3932	3933	3934	3935
7540	3936	3937	3938	3939	3940	3941	3942	3943
7550	3944	3945	3946	3947	3948	3949	3950	3951
7560	3952	3953	3954	3955	3956	3957	3958	3959
7570	3960	3961	3962	3963	3964	3965	3966	3967
7600	3968	3969	3970	3971	3972	3973	3974	3975
7610	3976	3977	3978	3979	3980	3981	3982	3983
7620	3984	3985	3986	3987	3988	3989	3990	3991
7630	3992	3993	3994	3995	3996	3997	3998	3999
7640	4000	4001	4002	4003	4004	4005	4006	4007
7650	4008	4009	4010	4011	4012	4013	4014	4015
7660	4016	4017	4018	4019	4020	4021	1022	4023
7670	4024	4025	4026	4027	4028	4029	4030	4031
7700	4032	4033	4034	4035	4036	4037	4038	4039
7710	4040	4041	4042	4043	4044	4045	4046	4047
7720	4048	4049	4050	4051	4052	4053	4054	4055
7730	4056	4057	4058	4059	4060	4061	4062	4063
7740	4064	4065	4066	4067	4068	4069	4070	4071
7750	4072	4073	4074	4075	4076	4077	4078	4079
7760	4080	4081	4082	4083	4084	4085	4086	4087
7770	4088	4089	4090	4091	4092	4093	4094	4095

Octal 7000-7777
Decimal 3584-4095

OCTAL TO DECIMAL CONVERSION, FRACTIONS

Octal	*Decimal*	*Octal*	*Decimal*	*Octal*	*Decimal*	*Octal*	*Decimal*	*Octal*	*Decimal*
.00000	.000000	.10000	.125000	.20000	.250000	.30000	.375000	.40000	.500000
.00100	.001953	.10100	.126953	.20100	.251953	.30100	.376953	.40100	.501953
.00200	.003906	.10200	.128906	.20200	.253906	.30200	.378906	.40200	.503906
.00300	.005859	.10300	.130859	.20300	.255859	.30300	.380859	.40300	.505859
.00400	.007812	.10400	.132812	.20400	.257812	.30400	.382812	.40400	.507812
.00500	.009765	.10500	.134765	.20500	.259765	.30500	.384765	.40500	.509765
.00600	.011718	.10600	.136718	.20600	.261718	.30600	.386718	.40600	.511718
.00700	.013671	.10700	.138671	.20700	.263671	.30700	.388671	.40700	.513671
.01000	.015625	.11000	.140625	.21000	.265625	.31000	.390625	.41000	.515625
.01100	.017578	.11100	.142578	.21100	.267578	.31100	.392578	.41100	.517578
.01200	.019531	.11200	.144531	.21200	.269531	.31200	.394531	.41200	.519531
.01300	.021484	.11300	.146484	.21300	.271484	.31300	.396484	.41300	.521484
.01400	.023437	.11400	.148437	.21400	.273437	.31400	.398437	.41400	.523437
.01500	.025390	.11500	.150390	.21500	.275390	.31500	.400390	.41500	.525390
.01600	.027343	.11600	.152343	.21600	.277343	.31600	.402343	.41600	.527343
.01700	.029296	.11700	.154296	.21700	.279296	.31700	.404296	.41700	.529296
.02000	.031250	.12000	.156250	.22000	.281250	.32000	.406250	.42000	.531250
.02100	.033203	.12100	.158203	.22100	.283203	.32100	.408203	.42100	.533203
.02200	.035156	.12200	.160156	.22200	.285156	.32200	.410156	.42200	.535156
.02300	.037109	.12300	.162109	.22300	.287109	.32300	.412109	.42300	.537109
.02400	.039062	.12400	.164062	.22400	.289062	.32400	.414062	.42400	.539062
.02500	.041015	.12500	.166015	.22500	.291015	.32500	.416015	.42500	.541015
.02600	.042968	.12600	.167968	.22600	.292968	.32600	.417968	.42600	.542968
.02700	.044921	.12700	.169921	.22700	.294921	.32700	.419921	.42700	.544921
.03000	.046875	.13000	.171875	.23000	.296875	.33000	.421875	.43000	.546875
.03100	.048828	.13100	.173828	.23100	.298828	.33100	.423828	.43100	.548828
.03200	.050781	.13200	.175781	.23200	.300781	.33200	.425781	.43200	.550781
.03300	.052734	.13300	.177734	.23300	.302734	.33300	.427734	.43300	.552734
.03400	.054687	.13400	.179687	.23400	.304687	.33400	.429687	.43400	.554687
.03500	.056640	.13500	.181640	.23500	.306640	.33500	.431640	.43500	.556640
.03600	.058593	.13600	.183593	.23600	.308593	.33600	.433593	.43600	.558593
.03700	.060546	.13700	.185546	.23700	.310546	.33700	.435546	.43700	.560546
.04000	.062500	.14000	.187500	.24000	.312500	.34000	.437500	.44000	.562500
.04100	.064453	.14100	.189453	.24100	.314453	.34100	.439453	.44100	.564453
.04200	.066406	.14200	.191406	.24200	.316406	.34200	.441406	.44200	.566406
.04300	.068359	.14300	.193359	.24300	.318359	.34300	.443359	.44300	.568359
.04400	.070312	.14400	.195312	.24400	.320312	.34400	.445312	.44400	.570312
.04500	.072265	.14500	.197265	.24500	.322265	.34500	.447265	.44500	.572265
.04600	.074218	.14600	.199218	.24600	.324218	.34600	.449218	.44600	.574218
.04700	.076171	.14700	.201171	.24700	.326171	.34700	.451171	.44700	.576171
.05000	.078125	.15000	.203125	.25000	.328125	.35000	.453125	.45000	.578125
.05100	.080078	.15100	.205078	.25100	.330078	.35100	.455078	.45100	.580078
.05200	.082031	.15200	.207031	.25200	.332031	.35200	.457031	.45200	.582031
.05300	.083984	.15300	.208984	.25300	.333984	.35300	.458984	.45300	.583984
.05400	.085937	.15400	.210937	.25400	.335937	.35400	.460937	.45400	.585937
.05500	.087890	.15500	.212890	.25500	.337890	.35500	.462890	.45500	.587890
.05600	.089843	.15600	.214843	.25600	.339843	.35600	.464843	.45600	.589843
.05700	.091796	.15700	.216796	.25700	.341796	.35700	.466796	.45700	.591796
.06000	.093750	.16000	.218750	.26000	.343750	.36000	.468750	.46000	.593750
.06100	.095703	.16100	.220703	.26100	.345703	.36100	.470703	.46100	.595703
.06200	.097656	.16200	.222656	.26200	.347656	.36200	.472656	.46200	.597656
.06300	.099609	.16300	.224609	.26300	.349609	.36300	.474609	.46300	.599609
.06400	.101562	.16400	.226562	.26400	.351562	.36400	.476562	.46400	.601562
.06500	.103515	.16500	.228515	.26500	.353515	.36500	.478515	.46500	.603515
.06600	.105468	.16600	.230468	.26600	.355468	.36600	.480468	.46600	.605468
.06700	.107421	.16700	.232421	.26700	.357421	.36700	.482421	.46700	.607421
.07000	.109375	.17000	.234375	.27000	.359375	.37000	.484375	.47000	.609375
.07100	.111328	.17100	.236328	.27100	.361328	.37100	.486328	.47100	.611328
.07200	.113281	.17200	.238281	.27200	.363281	.37200	.488281	.47200	.613281
.07300	.115234	.17300	.240234	.27300	.365234	.37300	.490234	.47300	.615234
.07400	.117187	.17400	.242187	.27400	.367187	.37400	.492187	.47400	.617187
.07500	.119140	.17500	.244140	.27500	.369140	.37500	.494140	.47500	.619140
.07600	.121093	.17600	.246093	.27600	.371093	.37600	.496093	.47600	.621093
.07700	.123046	.17700	.248046	.27700	.373046	.37700	.498046	.47700	.623046

Octal to Decimal Conversion, Fractions

Octal	*Decimal*	*Octal*	*Decimal*	*Octal*	*Decimal*
.50000	.625000	.60000	.750000	.70000	.875000
.50100	.626953	.60100	.751953	.70100	.876953
.50200	.628906	.60200	.753906	.70200	.878906
.50300	.630859	.60300	.755859	.70300	.880859
.50400	.632812	.60400	.757812	.70400	.882812
.50500	.634765	.60500	.759765	.70500	.884765
.50600	.636718	.60600	.761718	.70600	.886718
.50700	.638671	.60700	.763671	.70700	.888671
.51000	.640625	.61000	.765625	.71000	.890625
.51100	.642578	.61100	.767578	.71100	.892578
.51200	.644531	.61200	.769531	.71200	.894531
.51300	.646484	.61300	.771484	.71300	.896484
.51400	.648437	.61400	.773437	.71400	.898437
.51500	.650390	.61500	.775390	.71500	.900390
.51600	.652343	.61600	.777343	.71600	.902343
.51700	.654296	.61700	.779296	.71700	.904296
.52000	.656250	.62000	.781250	.72000	.906250
.52100	.658203	.62100	.783203	.72100	.908203
.52200	.660156	.62200	.785156	.72200	.910156
.52300	.662109	.62300	.787109	.72300	.912109
.52400	.664062	.62400	.789062	.72400	.914062
.52500	.666015	.62500	.791015	.72500	.916015
.52600	.667968	.62600	.792968	.72600	.917968
.52700	.669921	.62700	.794921	.72700	.919921
.53000	.671875	.63000	.796875	.73000	.921875
.53100	.673828	.63100	.798828	.73100	.923828
.53200	.675781	.63200	.800781	.73200	.925781
.53300	.677734	.63300	.802734	.73300	.927734
.53400	.679687	.63400	.804687	.73400	.929687
.53500	.681640	.63500	.806640	.73500	.931640
.53600	.683593	.63600	.808593	.73600	.933593
.53700	.685546	.63700	.810546	.73700	.935546
.54000	.687500	.64000	.812500	.74000	.937500
.54100	.689453	.64100	.814453	.74100	.939453
.54200	.691406	.64200	.816406	.74200	.941406
.54300	.693359	.64300	.818359	.74300	.943359
.54400	.695312	.64400	.820312	.74400	.945312
.54500	.697265	.64500	.822265	.74500	.947265
.54600	.699218	.64600	.824218	.74600	.949218
.54700	.701171	.64700	.826171	.74700	.951171
.55000	.703125	.65000	.828125	.75000	.953125
.55100	.705078	.65100	.830078	.75100	.955078
.55200	.707031	.65200	.832031	.75200	.957031
.55300	.708984	.65300	.833984	.75300	.958984
.55400	.710937	.65400	.835937	.75400	.960937
.55500	.712890	.65500	.837890	.75500	.962890
.55600	.714843	.65600	.839843	.75600	.964843
.55700	.716796	.65700	.841796	.75700	.966796
.56000	.718750	.66000	.843750	.76000	.968750
.56100	.720703	.66100	.845703	.76100	.970703
.56200	.722656	.66200	.847656	.76200	.972656
.56300	.724609	.66300	.849609	.76300	.974609
.56400	.726562	.66400	.851562	.76400	.976562
.56500	.728515	.66500	.853515	.76500	.978515
.56600	.730468	.66600	.855468	.76600	.980468
.56700	.732421	.66700	.857421	.76700	.982421
.57000	.734375	.67000	.859375	.77000	.984375
.57100	.736328	.67100	.861328	.77100	.986328
.57200	.738281	.67200	.863281	.77200	.988281
.57300	.740234	.67300	.865234	.77300	.990234
.57400	.742187	.67400	.867187	.77400	.992187
.57500	.744140	.67500	.869140	.77500	.994140
.57600	.746093	.67600	.871093	.77600	.996093
.57700	.748046	.67700	.873046	.77700	.998046

Octal	*Decimal*
.00000	.000000
.00001	.000030
.00002	.000061
.00003	.000091
.00004	.000122
.00005	.000152
.00006	.000183
.00007	.000213
.00010	.000244
.00011	.000274
.00012	.000305
.00013	.000335
.00014	.000366
.00015	.000396
.00016	.000427
.00017	.000457
.00020	.000488
.00021	.000518
.00022	.000549
.00023	.000579
.00024	.000610
.00025	.000640
.00026	.000671
.00027	.000701
.00030	.000732
.00031	.000762
.00032	.000793
.00033	.000823
.00034	.000854
.00035	.000885
.00036	.000915
.00037	.000946
.00040	.000976
.00041	.001007
.00042	.001037
.00043	.001068
.00044	.001098
.00045	.001129
.00046	.001159
.00047	.001190
.00050	.001220
.00051	.001251
.00052	.001281
.00053	.001312
.00054	.001342
.00055	.001373
.00056	.001403
.00057	.001434
.00060	.001464
.00061	.001495
.00062	.001525
.00063	.001556
.00064	.001586
.00065	.001617
.00066	.001647
.00067	.001678
.00070	.001708
.00071	.001739
.00072	.001770
.00073	.001800
.00074	.001831
.00075	.001861
.00076	.001892
.00077	.001922

APPENDIX H

ARITHMETIC FORMATS

The format for single precision numbers in twos complement fixed point notation is treated at the beginning of Chapter 2. Let us first discuss some further properties of such numbers here.

Zero is represented by a word containing all 0s. Complementing this number produces all 1s, and adding 1 to that produces all 0s again. Hence there is only one zero representation and its sign is positive. Since the numbers are symmetrical in magnitude about the single zero representation, all even numbers both positive and negative end in 0, all odd numbers in 1 (the number all 1s represents -1). But since there are the same number of positive and negative numbers and zero is positive, there is one more negative number than there are nonzero positive numbers. This is the most negative number and it cannot be produced by negating any positive number (its magnitude is one greater than the largest positive number and its octal representation is 100000).

If ones complements were used for negatives one could read a negative number by attaching significance to the 0s instead of the 1s. In twos complement notation each negative number is one greater than the complement of the positive number of the same magnitude, so one can read a negative number by attaching significance to the rightmost 1 and attaching significance to the 0s at the left of it (the negative number of the largest magnitude has a 1 in only the sign position). In a negative integer, 1s may be discarded at the left just as leading 0s may be dropped in a positive integer. In a negative fraction, 0s may be discarded at the right. So long as only 0s are discarded the number remains in twos complement form because it still has a 1 that possesses significance; but if a portion including the rightmost 1 is discarded, the remaining part of the fraction is now a ones complement.

The computer does not keep track of a binary point — the programmer must adopt a point convention and shift the magnitude of the result to conform to the convention used. Two common conventions are to regard a number as an integer (binary point at the right) or as a proper fraction (binary point at the left); in these two cases the range of signed numbers represented by a single word is -2^{15} to $2^{15} - 1$ or -1 to $1 - 2^{-15}$.

Double Precision Arithmetic. In a double length fixed point number, the second word is simply an extension of the magnitude part of the number. A double length number consists of two words concatenated into a 32-bit string wherein bit 15 of the high order word is the sign, and bits 14-0 of that word and bits 15-0 of the low order word are the magnitude in twos complement notation [*see the upper illustration on the next page*]. The high order part of a negative number is therefore in ones complement form unless the low order part is null (at the right only 0s are null regardless of sign). Hence in processing double length numbers, twos complement operations are usually confined to the low order parts, whereas ones complement operations are generally required for the high order parts. The address of a double length number is the address of its more significant word.

Suppose we wish to negate the double length number whose high and low order words respectively are in AX and AY. We negate the low order part, but we simply complement the high order part unless the low order part is zero.

```
      JC    AY,ETZ,ZERO ;Low part zero?
      RSC   AY,P1       ;No,negate low
      RSC   AX          ;Complement high
      JU    .+3
ZERO: RSC   AX,P1       ;Yes,negate high
```

Note that the magnitude parts of the sequence of negative numbers from the

$+262{,}146_{10}$ = $+2000002_8$ =

0 000 000 000 001 000	0 000 000 000 000 010
15 0	15 0

$-262{,}146_{10}$ = -2000002_8 =

1 111 111 111 110 111	1 111 111 111 111 110
15 0	15 0

Double Precision Fixed Point Format

$+153_{10}$ = $+231_8$ = $+.462\times2^8$ =

0 100 110 010 000 000	0 000 000 0	10 001 000
15 14 0	15 8	7 0

-153_{10} = -231_8 = $-.462\times2^8$ =

1 011 001 110 000 000	0 000 000 0	10 001 000
15 14 0	15 8	7 0

Floating Point Format

most negative toward zero are the positive numbers from zero upward. In other words the negative representation $-x$ is the sum of x and the most negative number. Hence in multiple precision arithmetic, low order words can be treated simply as positive numbers. In unsigned addition a carry indicates that the low order result is just too large and the high order part must be increased. We add the number in A and A+1 to the number in GR1 and GR2, with the result going to GR1 and GR2.

```
FOA   ADD
RR    GR2,AX
MR    A+1,AY
RR    AO,GR2
SFA   NOT AOV
RS    GR1,P1
NOP
RR    GR1,AX
MR    A,AY
RR    AO,GR1
```

In twos complement subtraction a carry should occur unless the subtrahend is too large.

Floating Point Arithmetic

Software is available for processing floating point numbers. For a given word length, floating point format sacrifices some precision for a much greater range in order of magnitude. The software interprets the two-word floating point representation of a number as containing a sign, a 23-bit proper fraction, and an 8-bit exponent [*lower illustration above*]. The sign is bit 15 of the high order word, and in a positive number it is 0. The contents of bits 14-0 of the high word and bits 15-8 of the low word are interpreted only as a binary fraction; and the contents of bits 7-0 of the low word are interpreted as an integral exponent in excess 128 (200_8) code. Exponents from -128 to +127 are therefore represented by the binary equivalents of 0 to 255 (0-377_8). The negative of a number is represented by taking the twos complement of the sign and fraction only — the exponent is left in positive form. Zero is represented by all 0s in sign, fraction and exponent. The routines always represent a zero result in this form, but they in-

terpret any operand with a zero fractional part as being zero.

Most routines assume that all nonzero operands are normalized, and they normalize a nonzero result. A floating point number is considered normalized if the magnitude of the fraction is greater than or equal to ½ and less than 1; in other words the sign and the most significant bit of the fraction differ or the fraction is $-\frac{1}{2}$. These numbers thus have a fractional range in magnitude of ½ to $1 - 2^{-23}$ and an exponent range of -128 to +127. This corresponds to a decimal range of approximately 5×10^{-40} to 5×10^{38}. In some cases a routine may not give the correct result if the programmer supplies an operand that is not normalized.

Numbers of greater precision are produced simply by inserting words containing sixteen fraction bits between the words described above. As in the fixed point case, the address of a multiple precision number is the address of its highest order word.

GRI Computer Corporation

76 ROWE STREET, NEWTON, MASSACHUSETTS 02166

(617) 969-7346

20-40-001-A

1269.5000