INTRODUCTION TO PROGRAMMING

ON THE NOVA COMPUTER

by

CLEMENT C.Y. LAM, B.ENG. (McMaster)

PART A: OFF-CAMPUS PROJECT*

A project report submitted in partial fulfillment of the
requirement for the degree of
Master of Engineering

Dept. of Engineering Physics
McMaster University
Hamilton, Ontario

September, 1977

*One of two project reports: The other part is designated
PART B: McMASTER (On-Campus) PROJECT.

MASTER OF ENGINEERING (1977) MCMASTER UNIVERSITY

Department of Engineering Physics Hamilton, Ontario

TITLE: INTRODUCTION TO PROGRAMMING ON THE NOVA COMPUTER

AUTHOR: CLEMENT C.Y. LAM, B. Eng. (McMaster)

SUPERVISOR: Dr. T.J. Kennett

NUMBER OF PAGES: iv, 73

Abstract

A guide to programming the Nova computer is presented in this manual. Programming fundamentals, structured programming, testing and debugging and interrupt programming technique are also included.

ACKNOWLEDGEMENTS

I wish to thank my supervisor, Dr. T.J. Kennett, for his valuable guidance and suggestions throughout the course of this project. I also wish to thank Mr. K. Chin for the helpful discussions I had with him on many occasions. Finally, a sincere thanks to Dr. A. Robertson and Mr. N. Barkman for their proofreading of the manual.

TABLE OF CONTENTS

To my parents

Introduction

This manual is intended to give the student an introduction to programming of the Nova computer. Students who are not familiar with number systems, binary arithmetic and Nova computer architecture will find "Introduction to the Data General Nova"[2] helpful.

A little knowledge of computer architecture in general will surely augment the understandability of the manual. Finally, every term being asterisked in the following chapters is included in the glossary. (Appendix B of the manual).

Chapter 1 Programming Fundamentals[1, 4, 5]

Section 1.1 Procedure for writing computer programs*

There are four basic steps in writing programs.

1. Problem identification.
2. Formulation of an algorithm for solving the problem.
3. Implementation into the corresponding computer language.*
4. Testing and debugging* the program.

1.1.1 Problem identification

This step is very important. Failure to identify the problem correctly may result in a working program, yet it may not yield the solution to the problem. In this case, all the time used will have been wasted in implementing an "incorrect" program.

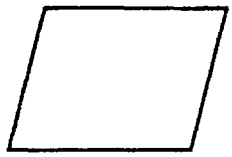
1.1.2 Formulation of an algorithm* for solving the problem

Flowcharting* is one of the most important steps in the design of a program. In essence, flowcharts are pictorial representations of algorithms providing the programmer with an indication of the flow of the program. They are a combination of symbols with English and mathematical statements that clearly defines every step in solving the problem. There are a number of symbols, each of which indicates the type of operation to be performed. More detailed explanations of the operation are contained within the symbol.

The following are some of the symbols most often used.

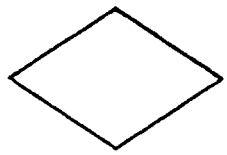
Terminal Symbol: start, stop, halt, delay or interrupt.

Input/Output Symbol: Input of information to the computer, output of information from the computer.



Process Symbol: processing of information.

Decision Symbol: Decision resulting in a number of alternate paths.



Connector Symbol: A junction connector (exit to, entry from another part of chart).

Flow Direction Symbols:

1.1.3 Implementation into the corresponding computer language

After a flowchart has been drawn, the computer program can be implemented by translating the operation within each symbol into the corresponding computer language instructions.

1.1.4 Testing and debugging the program

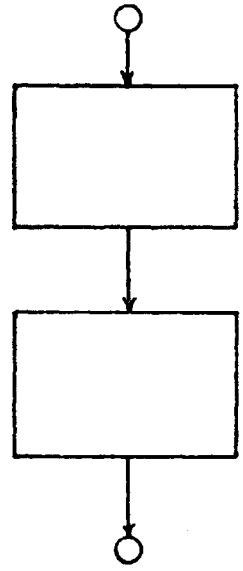
Before the program can be accepted as a valid solution to the problem, it should be subjected to tests. Failure to pass the prescribed tests indicates that errors or "bugs" exist within the program. After "debugging", the program should be re-tested. The testing and the debugging process is repeated until the program is essentially free from any detectable error.

Section 1.2 Structured Programming*

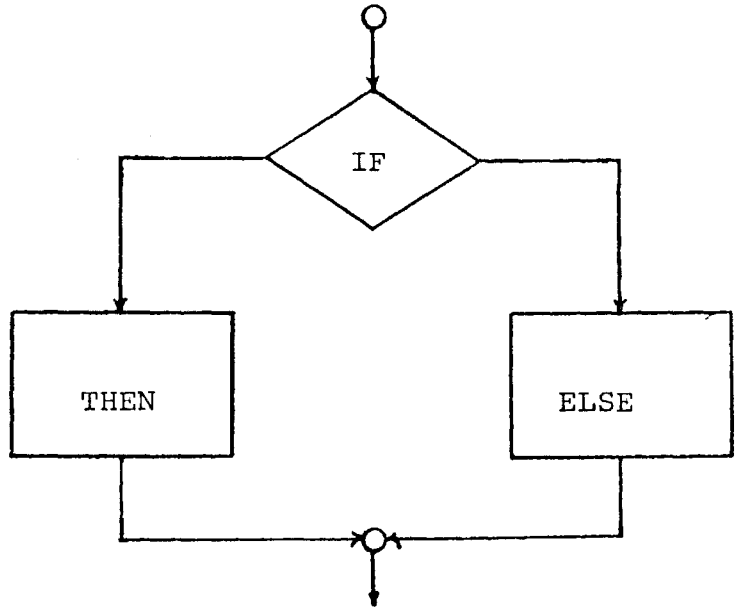
1.2.1 Programming structures

There are basically three kinds of programming structures.

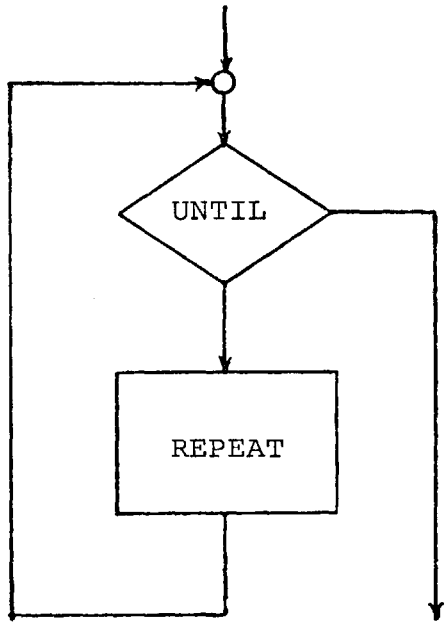
1. Simple sequence

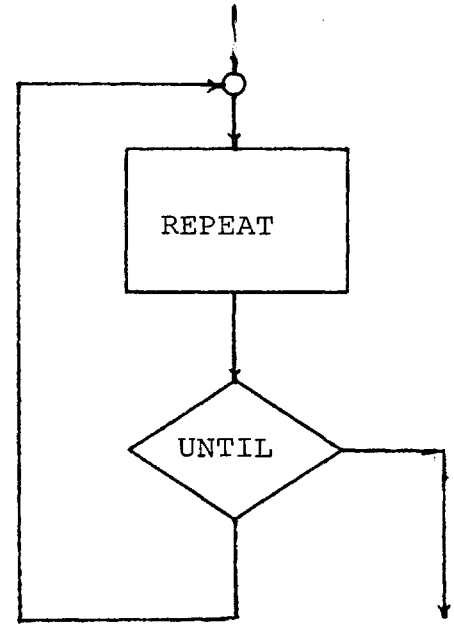


2. Selection



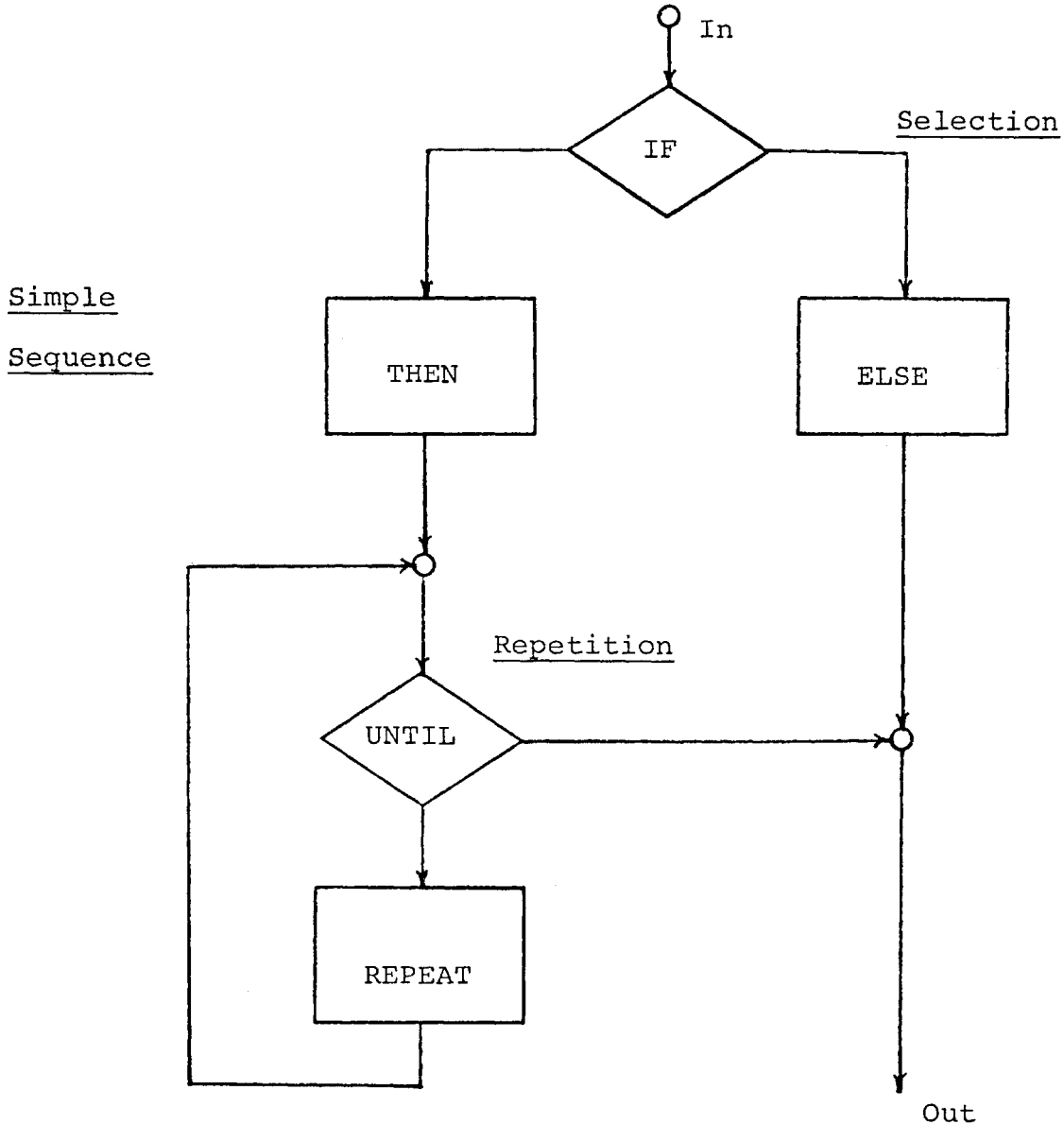
3. Repetition





Any kind of processing, combination of decisions and logic can be accomodated with one of these control structures or a combination of them. Each control structure is characterized by a simple and single point of transfer of control into the structure, and a single point of transfer of control out of the structure. This is the important concept of SINGLE ENTRY/SINGLE EXIT. These control structures can be nested, as shown in Figure 1.1, but they retain their characteristic of single entry/single exit.

Figure 1.1 Nesting of control structures



1.2.2 Branching*

In debugging a program, a lot of time is often wasted in tracing the flow of the program. If the flow is in one direction, for example, from top to bottom, then effort and time will be saved in tracing the program.

The following points are useful when writing programs which flow in one direction.

1. The three control structures discussed in 2.1 can be combined to form a program so simple that the control will flow from top to bottom. There should be a minimum of back-tracking within the program.

2. Attention should be paid to branching within a program. A program may become very complex to follow if it contains too many branches. Whenever it is possible, branching should be kept to a minimum.

1.2.3 Modular programming*

When one starts to write long programs it becomes increasingly difficult to check and debug these programs. It is in this context that modular programming becomes important.

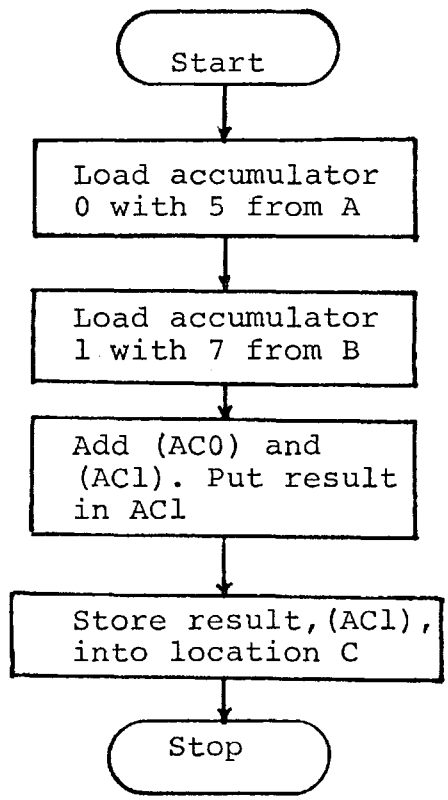
The concept is very basic. Since small programs are easy to check and debug, any long programs should be made up of a number of small sub-programs or modules. The modules can be completely checked before being linked together into a complete program.

Chapter 2 A simple programming example using Nova Computer[2,5]

Problem: Write a program which will add 5_8 and 7_8 together and store the result in a memory location*. The two octal numbers are retrieved from two memory locations.

Name of Location	Content
A	5_8
B	7_8
C	result of addition

The following is a flowchart solution to the problem.



AC0 means accumulator 0

(AC1) means the content of AC1

The computer interprets commands which must be coded in binary* notation. A language which is represented by binary codings is called machine language*. The flowchart alogorithm has to be implemented in the corresponding machine language before the computer can execute it. The following is a programming solution to the problem in machine language.

Memory Location	Content in Octal* (machine codings)	Meaning
100	020105	load (105) into AC0
101	024106	load (106) into AC1
102	107000	(AC1) - (AC0) _ (AC1)
103	044107	store result in location 107
104	063077	stop execution
105	000005	5_8 is in this location
106	000007	7_8 is in this location
107	000000	reserved location which receives the result of addition

One should observe that machine instructions do not mean much to the programmer. In order to understand a program in machine language, one has to find the corresponding meaning of each machine instruction. It is in this context that assembly language* is used instead of machine language. An assembly language is merely a mnemonic* equivalence of the machine language. Its purpose is to increase the understandability of a program by a programmer. The following is the same program written in assembly language.

Location	Machine Code (in octal format)	Mnemonics	Comment
		.LOC 100	; specify where to put the pro\ram in ; memory
100	020105	LDA 0, A	; load AC0 with the content of A
101	024106	LDA 1, B	; load AC1 with the content of B
102	107000	ADD 0, 1	; (AC1) = (AC0) + (AC1)
103	044107	STA 1, C	; store (AC1) into location C
104	063077	HALT	; stop program execution
105	000005	A: 5	; 5 is stored here. A label A is given to this location
106	000007	B: 7	; (B) = 7
107	000000	C: 0	; Reserved location for storing the result. Its label is C
		.END	; end of program

After execution of this program, C will contain 14_8.

A few things are worth mentioning here:

1. .LOC n and .END are pseudo-operations*. Pseudo operations are not instructions to be performed but directives to the assembler*. .LOC n specifies the starting address of the program in memory. .END is an indication of the end of a program.
2. A program must have a HALT instruction to stop execution. If there is not a HALT instruction after STA 1, C, the program will continue to run, causing constants and data to be executed as instructions. Care should be taken to

always avoid this situation.

3. The following locations are reserved for special purposes.

   a. location 20-27 auto-incrementing* when be ng indirectly addressed*.

   b. location 30-37 auto-decrementing* when being indirectly addressed.

   c. location 0 reserved for interrupt programming*.

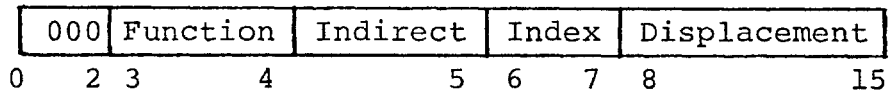
4. Anything after a semi-colon in a program will be regarded as a comment by the Nova assembler. A maximum of five characters can be used for labels which are delimited by a colon. Within an instruction, either a comma or space can be used as a delimiter. (Refer to the example above).

Chapter 3 Programming with the Nova[1, 2, 5]

Section 3.1 Addressing modes (Memory Reference Instructions)*

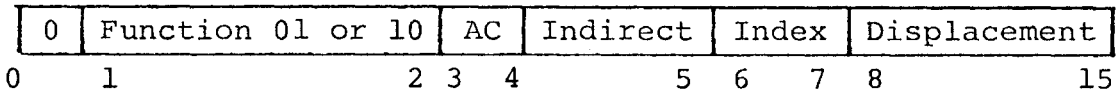
The memory reference instructions (MRI) interact with memory or the program counter (PC). They are specified by (placing) a 0 in bit 0 and either 00 (referenced without AC), 01 or 10 (referenced with AC) in bit 1-2.

MRI without AC



Function Field	Instruction	Meaning
00	JMP	JUMP
01	JSR	Jump to SubRoutine
10	ISZ	Increment and Skip on Zero
11	DSZ	Decrement and Skip on Zero

MRI with AC



Function Field	Instruction	Meaning
01	LDA	LoaD Accumulator
10	STA	STore Accumulator

AC can be 00, 01, 10 and 11 referring to AC0, AC1, AC2 and AC3 respectively. Each 16 bit MRI instruction word is divided into four fields:

	bits
command field (C)	0 through 4
addressing mode field (I)	5
index field (X)	6 through 7
displacement field (D)*	8 through 15

The command field determines the type of instructions: more data; modify memory; jump.

Every MRI must contain an effective address (E) which specifies which memory cell is to be referenced.

The effective address (E) is formed by the index X and the displacement D. The index X refers to a register or accumulator to whose content is added the displacement D, resulting in the address of the desired memory cell.

$$E = (X) + D$$

Move Data MRI's

	LDA AC, D, X	loads accumulator AC with the contents
or	LDA AC, NAME†	of the memory cell specified by the effective address E, which is made up of the displacement D and index X.
	STA AC, D, X	stores the contents of accumulator
or	STA AC, NAME	AC into memory location E.

Modify Memory MRI's

	ISZ D, X	increments (E) by 1 and stores it back
or	ISZ NAME	into memory location E. If new (E) = 0, the next instruction is skipped. If (E) $\neq$ 0, the nominal program sequence is followed.
	DSZ D, X	decrements (E) by 1 and skips if
or	DSZ NAME	new (E) = 0.

Jump MRI's

	JMP D, X	loads E into PC, take the next instruction
or	JMP NAME	from location E, and continues sequential operation from there, i.e., transfers control to memory location E.
	JSR D, X	first computes E and saves (PC) + 1 into
or	JSR NAME	AC3, then loads E into the PC, transfers control to memory location E.

†NAME is the label given to the effective address.

3.1.1 Direct addressing* (I=0)

There are four kinds of direct addressing modes.

(a) Page zero* addressing (X=00)

$E = 0 \qquad 0 \leq D \leq 377_8$

E can be referenced from anywhere in memory.

Data should be placed in page 0 in order that they may be accessed by various parts of a program.

e.g., 3.1.

If location 100 contains 001234 and is labelled A, then a LDA 0, A instruction will load 001234 into AC0 no matter where in memory this instruction resides. It can be done as long as A is inside page 0.

(b) Relative addressing (X=01)

$E = \text{current (PC)} + D \qquad -200_8 \leq D \leq 177_8$

e.g., 3.2.

(PC)	Machine code	Mnemonic	Comments
		.LOC 625	
625	020410	LDA 0, A	; load AC0 with content of A
		========	
		========	
635	001234	A: 001234	
		.END	

When LDA 0, A is executed, (PC) = 625 and A is at location 635. Relative addressing will be assumed if A is outside page 0 by the assembler and the created instruction to load (A) = 001234 into AC0 will results.

$E = (PC) + D = 625 + 10 = 635$

(c) Base register addressing (X=10)

$E = (AC2) + D \qquad -200_8 \leq D \leq 177_8$

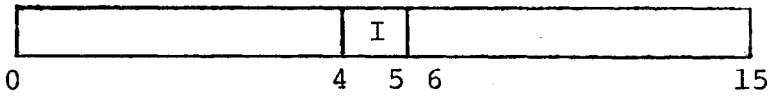
e.g., 3.3.

```
Location  Machine codes   Mnemonics          Comment
                          .LOC 40
    40        010000      A:  010000
                          .LOC 100
   100        030040      LDA 2, A       ; page zero addressing.
                                             Load (A)
                                          ; into AC2
   101        021000      LDA 0,0,2      ; Base register addressing
                                             with
                                          ; respect to (AC2).
                                          ; load AC0 with content
                                             of
                                          ; address pointed by
                                             (AC2) + D.
                          --------       ; In this case (10000) =
                                             001234
                          ---------      ; will be loaded into AC0.
                          .LOC 10000
 10000        001234      001234
                          ---------
                          .END
```

(d) Base register addressing (X=11)

$E = (AC3) + D \qquad -200 \leq D \leq 177$

3.1.2 Indirect Addressing (I=1)

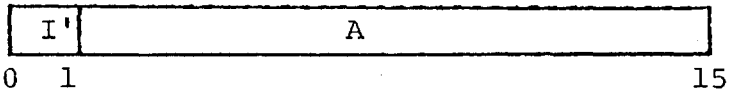


This mode of addressing is determined by I=1. It will be used in conjunction with the four addressing modes discussed before. By using indirect addressing, each memory cell in a 32K (77777_8) configuration can be referenced.

The meaning of E now differs from the previous examples. In the direct addressing mode, E corresponds to the effective address in which the operand is contained. In the indirect addressing mode, E no longer refers to the address of the operand, but it will contain an address pointing to the operand.

address of operand = (E) $\quad 0 \leq (E) \leq 77777$

Many levels of indirect addressing are possible by the use of a 1 in bit 0(I') of the referenced word.



I' = 1 means A is an address of the operand.

I' = 0 means A is the operand.

e.g., 3.4

If location 200 contains 000123

and location 123 contains 001234

then LDA 0, @ 200 will load 001234 into AC0.

@ is the symbol for indirect addressing.

e.g., 3.5.

This example will show how operands outside page zero may be referenced by using indirect addressing.

If location 100 contains 100123

location 123 contains 010000

location 10000 contains 001234

then LDA 0, @ 100 will load 001234 into AC0.

There are two levels of indirect addressing involved. The first level is indicated by the @ in the instruction. The second level is specified by a 1 in bit 0 in the referenced word, that is the content of location 100. Since the indirect bit is not set in location 123, location 010000 will contain an operand to be fetched.

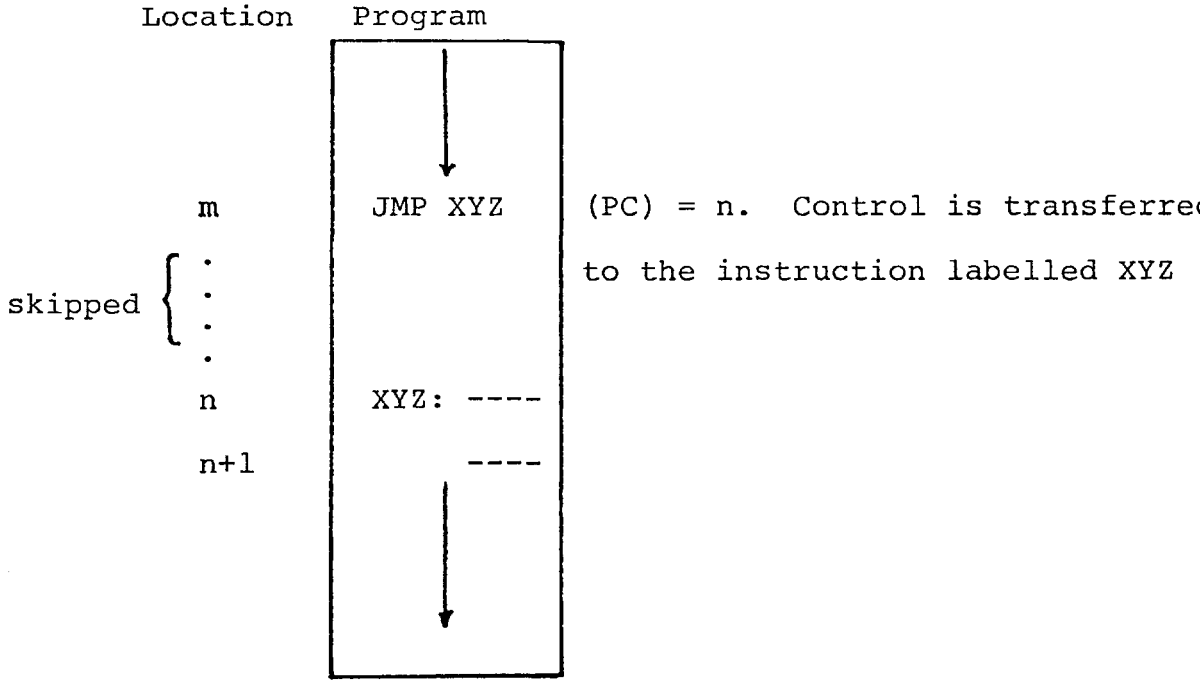
Question: What would happen if one had in location 100 → 100100_8 and the instruction LDA 0 @ 100 is executed?

Answer: The computer would be fetching loc. 100 forever and never get the number in AC0. The only way to stop the Nova would be to turn power off. (One would normally think the computer will stop by pressing STOP, but it does not work in this case).

3.1.3 Jump MRI's in branching

The JSR instruction will be explained in section 3.3. The JMP - "JUMP" - instruction is used specifically to alter the flow of a program. The program which is stored in memory is normally executed sequentially, since the program counter is incremented by 1 following execution of an instruction.

It may be desirable, at some point in a program, to branch to another group of instructions which resides somewhere else in memory. To perform this branching, the memory address where the new block of instructions begins must be put into the PC.



JMP instructions are used for unconditional branchings.

3.1.4 Modify Memory MRI's in looping

ISZ and DSZ can be used to implement the repetition programming structure.

e.g., 3.6.

Write a program to execute the routine "LOOP" ten times.

```
          .LOC 100
START:    LDA 0,CONST       ;INITIALIZE COUNTER
          STA 0,COUNT

LOOP:     -----
          -----             ;EXECUTES THIS 10 TIMES
          -----
          DSZ COUNT         ;DECREMENT (COUNT),SKIP IF ZERO
          JMP LOOP
          HALT

CONST:    12
COUNT:    0
          .END
```

The value of COUNT is initialized to 12_8 each time the program is executed.

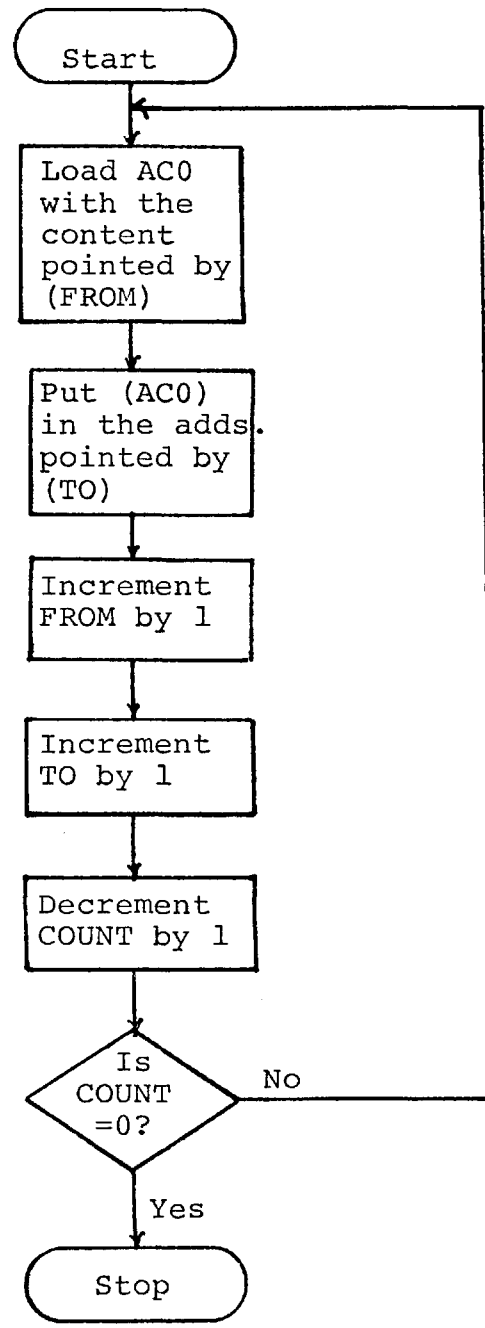
ISZ could have been used instead of DSZ. In this case CONST should have been assigned the value of -12_8, in 2's complement* form, at the start of the program.

3.1.5 Block transfer program

Problem: Write a program which will move a block of data from one place in memory to another. Indirect addressing will be used to solve the problem.

The starting address of the block to be moved is stored in a location labelled FROM and the starting address of the new block is stored in TO. COUNT contains the number of words to be moved.

The following is a flow chart solution.



The following is an assembly program of the solution.

```
;       BLOCK TRANSFER PROGRAM
;       THIS PROGRAM WILL TRANSFER A BLOCK OF INFORMATION
;       FROM ONE PLACE TO THE OTHER INSIDE THE CORE

        .LOC 100

START:  LDA 0, .FROM    ;INITIALIZE FROM, TO & COUNT
        STA 0, FROM
        LDA 0, .TO
        STA 0, TO
        LDA 0, CONST
        STA 0, COUNT
;
LOOP:   LDA 0,@FROM     ;(AC0)=WORD TO BE MOVED
                        ;(FROM)= ADDS OF THE WORD TO BE MOVED
        STA 0,@TO       ;THE WORD IS STORED IN THE NEW
                        ;LOCATION SPECIFIED BY (TO)
        ISZ FROM        ;GENERATE THE ADDS OF NEXT WORD TO
                        ;BE MOVED
        ISZ TO          ;GENERATE THE ADDS OF NEXT AVAILABLE
                        ;LOCATION IN THE NEW BLOCK
        DSZ COUNT       ;TEST IF ALL WORDS HAVE BEEN MOVED.
                        ;SKIP NEXT INSTRUCTION IF COUNT=0.
        JMP LOOP        ;NOT YET, GO BACK TO LOOP
        HALT            ;END OF JOB.
;
;       DATA & CONSTANTS
;
.FROM:  200             ;200 OCTAL IS THE ADDS OF THE FIRST
                        ;WORD TO BE MOVED
FROM:   0
.TO:    1000            ;1000 OCTAL IS THE STARTING ADDS OF
TO:     0               ;THE NEW BLOCK
CONST:  10              ;THE BLOCK TO BE MOVED CONTAINS 10 OCTAL
COUNT:  0               ;WORDS
;
;       BLOCK OF INFO. TO BE MOVED
;
        .LOC 200
        1
        2
        3
        4
        5
        6
        7
        10
        .END
```

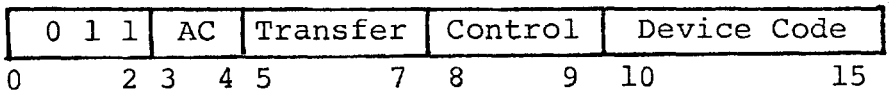
After the execution of the program, locations 1000 through 1010 will contain 1, 2, 3, 4, 5, 6, 7 and 10.

Suggested exercises

Appendix A, No. 182.

Section 3.2 Input/Output Instructions (I/O)

The I/O instructions control all the operations between the processing unit and peripheral equipment*. Each Nova I/O instruction has the following format:



Data Transfer

Any transferring of data is done between a selected device and a selected accumulator. The accumulator involved is specified by bits 3 and 4 (0, 1, 2, 3). The device involved is specified by the device code in bits 10 through 15. Bits 10 through 15 decode to 64 unique possibilities; however, only 62 devices may be addressed (01 through 76_8). Device code 00 is not used, and 77 is a special function code denoting the CPU*. Within a device, there may exist up to 3 data buffers* (A, B and C). Bits 5 through 7, the transfer field, specify the buffer involved and the direction of the data transfer, whether IN or OUT. An IN transfer implies a data transfer from the device buffer to the processor. An OUT transfer implies a data transfer from the processor to the device buffer.

If the transfer field is	The transfer is	The mnemonic is
0	No I/O transfer	NIO
1	Data IN from buffer A	DIA
2	Data OUT to buffer A	DOA
3	Data IN from buffer B	DIB
4	Data OUT to buffer B	DOB
5	Data IN from buffer C	DIC
6	Data OUT to buffer C	DOC
7	(reserved for slap tests described later)	

Control field

Once the device, buffer and accumulator have been selected, it is necessary to specify control information to the device via the control field, bits 8 and 9.

Associated with every device are two flip-flops* or flags. Busy and Done. If both flags are clear (reset), the device is in the idle mode. To palce the device into operation, the Busy flag must be set. After the device has processed the unit of data on a Data Out instruction, or when a device has information available in a buffer register

on a Data In instruction, the Busy flag is cleared and the Done flag is set.

The combination of these two flags produces four possibilities, each representing a state of the device.

Busy	Done	Device is
0 (clear)	0 (clear)	idle - not in use
1 (set)	0 (clear)	busy - it is in the process of performing some operation - it is not available for service at this time
0 (clear)	1 (set)	done - it has completed a specific operation
1 (set)	1 (set)	(not normally a valid condition)

Using the control field of the I/O instruction, the following control functions can be specified by appending the appropriate mnemonic to the instruction.

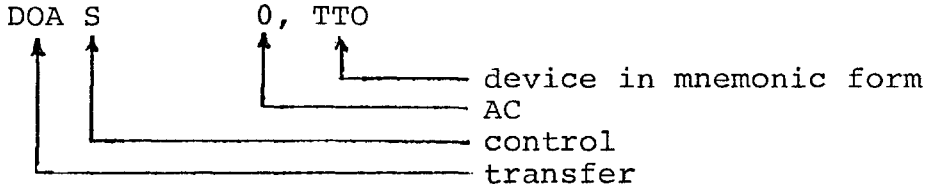
If the mnemonic is	The control function is
- (nothing)	no control
S	set the Busy flag and clear the Done flag. Thus starting the device.
C	Clear both the Busy and Done flags, thus idling the device.
P	special purpose output for customer applications.

The general format of an I/O instruction is

Transfer (Control) AC, Device code.

Example 3.7

To type the character contained in AC0 on th teletype



This instruction causes the contents of AC0 to be transferred to Buffer A of the teletype* output (TTO), the TTO is then started by the S in the control field, and the character is typed. The S pulse will set the Busy flag for the length of time necessary to type a character.

Special functions

Using the special function transfer code 7, it is possible to test the status of the Busy and Done flags and to conditionally skip the next instruction as a result of the test.

Mnemonic	Transfer code	Control code	Function
SKPBN	7	0	skip the next instruction if the Busy flip-flop is zero.
SKPBZ	7	1	skip the next instruction if the Busy flip-flop is zero.
SKPDN	7	2	skip if the Done flip-flop is non-zero.
SKPDZ	7	3	skip if the Done flip-flop is zero.

Each skip-on-flag instruction must specify a specific device.

3.2.1 Teletype Output (TTO)

For the Nova, the teletype (TTY) is an important I/O device. While the TTO can only handle 10 characters/sec, the computer is capable of executing an instruction in 2 μsec. Because of the mismatch of speed between the computer and the TTO, proper synchronization has to be maintained when using the TTO. The most common procedure is to test the status of the associated Busy and Done flags in the TTO.

Caused by	Busy	Done	TTO is
IORST (I/O reset) or a C in the control field or manual reset	0	0	idle
an S in the control field	1	0	typing a character
completion	0	1	ready to type another character

From the table above, in order to check if the TTO is ready for service, it is sufficient to test the status of the Busy flag. Busy = 1 indicates that the TTO is not yet ready while Busy = 0 means the TTO is ready.

Ascii code*

In computer I/O, information is frequently encoded in a seven-bit format known as the American Standard Code for Information Interchange (Ascii). When set into operation, the TTY printer will decode the incoming Ascii coded character

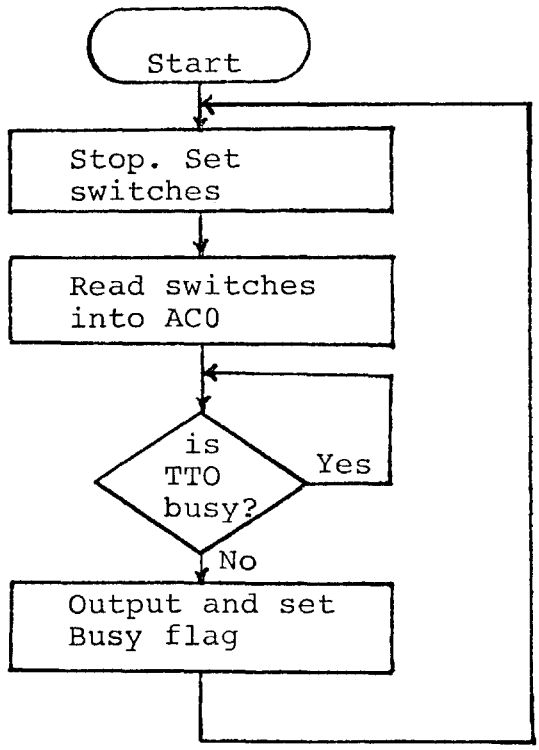
and type it out in the normal form. When the TTY is used as an input device, all information will be encoded before transmission to the computer.

e.g., 3.8

Character	Ascii code
1	060_8
A	101_8
CR (carriage return)	015_8

e.g., 3.9

Write a program which will read the Ascii codings from switches and output the corresponding character to the TTO.



```
;       THIS PROGRAM WILL READ THE ASCII CODINGS FROM THE
;SWITCHES AND OUTPUT THE CORRESPONDING CHARACTER TO THE
;TTO
;
        .LOC 100Ø
START:  HALT                ;SET SWITCHES. PRESS CONTINUE TO
                            ;CONTINUE
        READS Ø             ;READ SWITCHES INTO ACØ
        SKPBZ TTO           ;TEST TO SEE WHETHER TTO IS BUSY
        JMP .-1             ;YES, KEEP TESTING
        DOAS Ø,TTO          ;NO, OUTPUT CHARACTER AND SET BUSY=1
        JMP START           ;GET ANOTHER CHARACTER
        .END
```

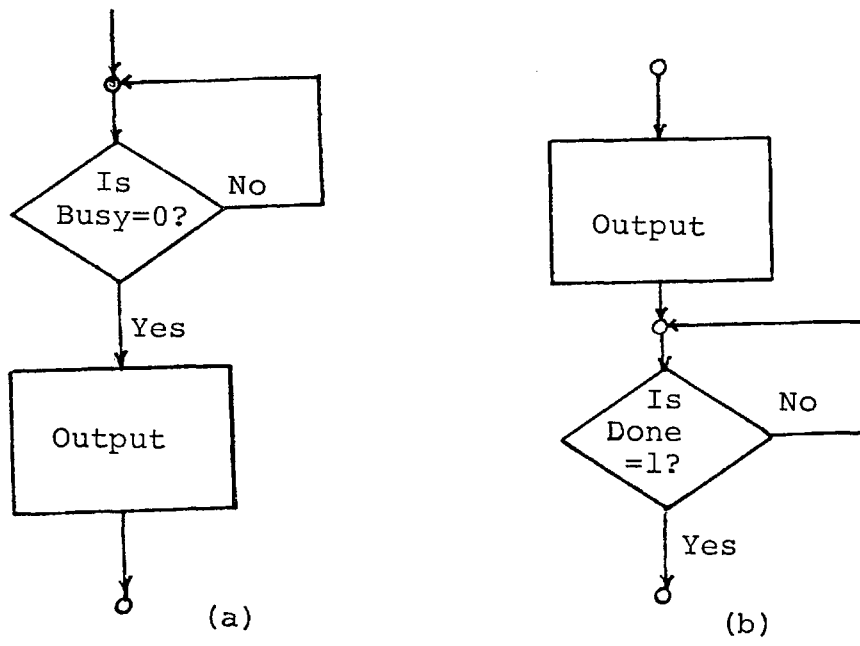
Notes: (1) READS AC is an input instruction which uses a device code of 77 (77 ≡CPU).

READS AC ≡ DIA AC, CPU

(2) .-1 means current location minus one.

There are basically two procedures for using the skip instructions in a loop to process a series of characters.

Consider the following two cases:



Case (a) is the more efficient, since all of the waiting time can be utilized by the program for other processes. Any useful time for computation will be wasted in the waiting loop in case (b).

3.2.2 Teletype Input (TTI)

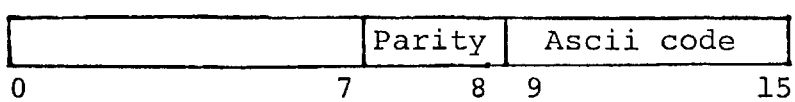
The TTO and TTI are two separate devices, each having its own interface and device code. The TTI also has two flags. Busy and Done. One can determine the status of the TTI by checking these two flags.

Caused by	Busy	Done	TTI action
IORST or a C in the control field or manual reset	0	0	idle
a S in the control field	1	0	A character is requested by the paper tape reader
completion	0	1	A character has been received by the paper tape reader or from the keyboard

A key has to be struck before Done will go to 1. By testing the Done flag, one will know if there is a character ready to be accepted.

Parity* bit for error-checking

On TTI input, besides the 7 bits ascii coding of the character, an extra bit called the parity bit will come together with the ascii code to make up a 8-bit character. The character is then put into bit 8 through 15 of the specified AC.



parity bit (bit 8) = 1 if odd number of 1's occur in bit 9 through 15.

= 0 if even number of 1's occur in bit 9 through 15.

e.g., 3.10

Write a program which will read in ascii characters from the TTI keyboard.

```
;       THIS PROGRAM WILL READ IN ASCII CHARACTERS FROM
;THE TTI KEYBOARD
;
        .LOC 100
;
START:  SKPDN TTI       ;WAIT UNTIL A KEY IS STRUCK
        JMP .-1
        DIAC 0,TTI      ;BRING THE CHARACTER IN AND CLEAR DONE FLAG
        LDA 1,MASK      ;MASK OUT THE 7-BIT ASCII CODE INTO
        AND 1,0         ;AC0
;       -------
;       -------         COMPUTATION
;       -------
        JMP START       ;GET ANOTHER CHARACTER
MASK:   177
        .END
```

In this example, the parity bit of the ascii code is set to zero by ANDing the entire word with a mask 177_8.

Masking is a technique of information extraction. If one wants to retain certain bits of an accumulator while zeroing the rest, one can perform a masking operation by ANDing the number with a second number which contains 1's in the bits of the original number that are to be preserved.

e.g., 3.11

if [AC0] = 303 ascii of C

[AC1] = 177 mask

then And 1, 0 will put 103 into AC0.

$303_8 \equiv 11000011_2$

And $177_8 \equiv 01111111_2$

$103_8 \equiv 01000011_2$

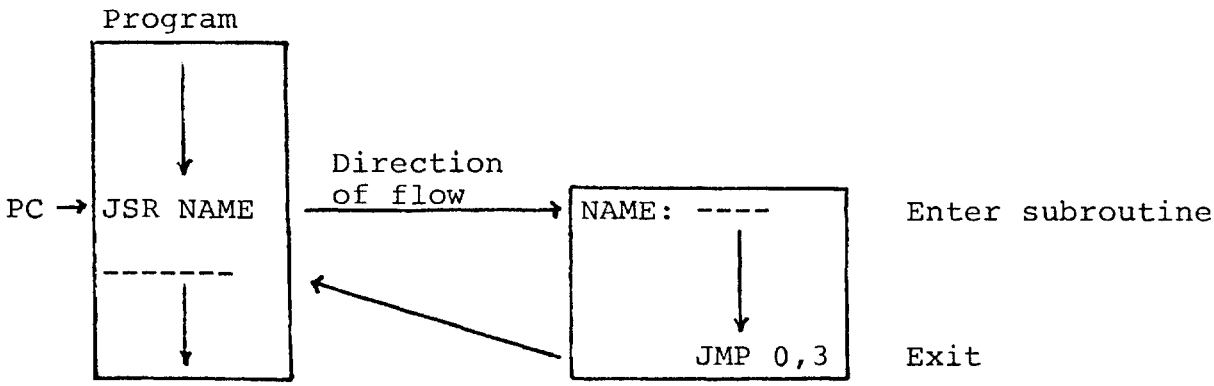
Suggested exercises

Appendix A, Nos. 3, 4 and 5.

Section 3.3 Subroutines*

The JSR NAME - "Jump to subroutine labelled NAME" - provides branching similar to that of the JMP instruction; the main difference between the JMP and MSR instruction is that the JSR instruction not only branches to some other group of instructions, but it also retains the memory address where it jumped from. This feature is extremely useful when writing groups of instructions which will be performed many times in a program and implementing structured programs.

When a JSR NAME instruction is encountered during the course of program execution, the program counter is incremented and loaded into AC3, the address of NAME will then be placed into the PC, thus effecting transfer of control to the subroutine. A JMP 0,3 will return control back to the calling program when this is desired.



Before entering	After exiting
(AC3) := current(PC)+1	
(PC) := adds. of NAME	(PC) := (AC3)

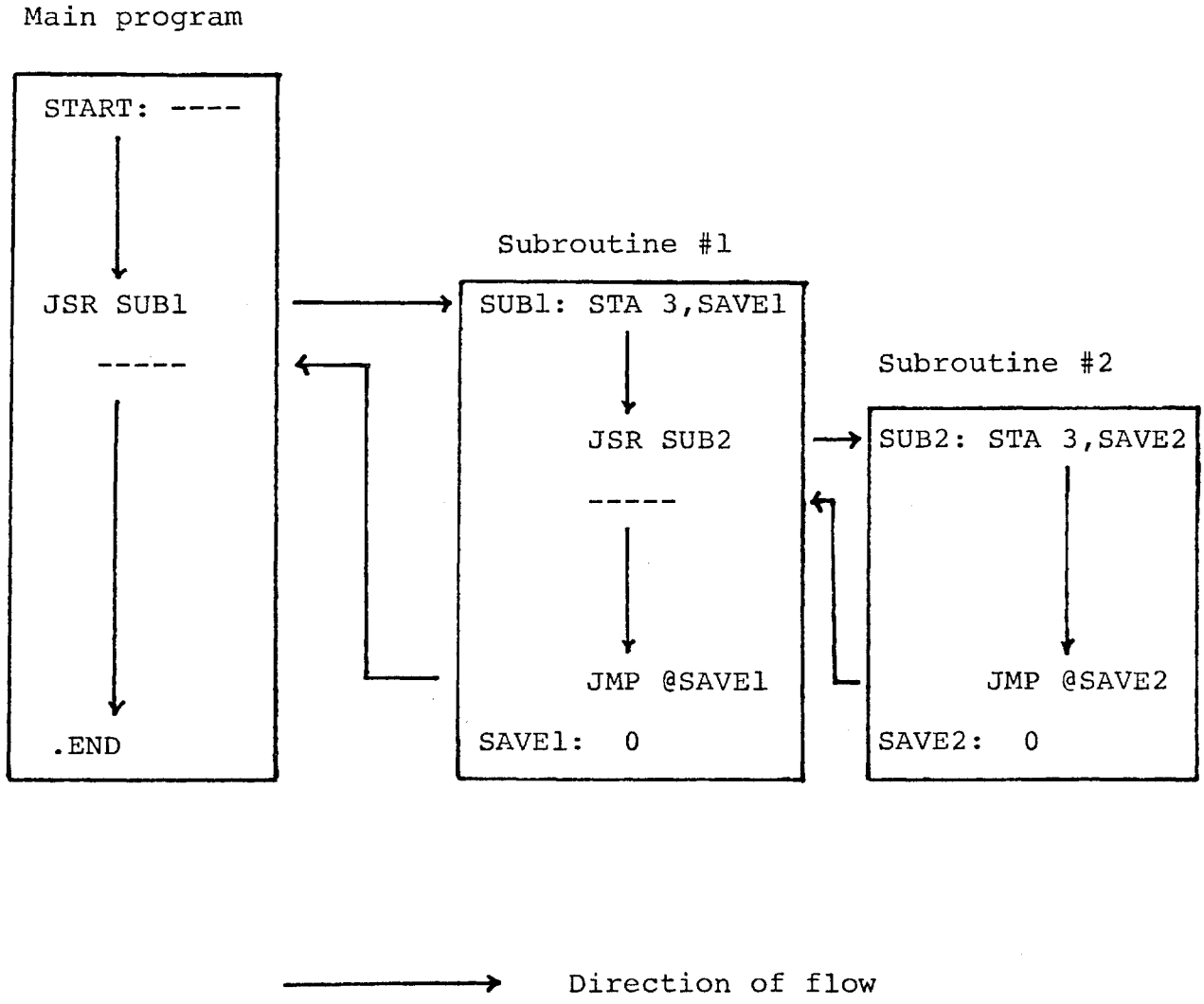
When executing a subroutine, AC3 is occupied by the return address. It is good practice to save the return address in a memory location so that AC3 is available for other purposes.

Subroutine calls within a subroutine are made possible by saving each of the return addresses. An indirect jump to the saved address will return control to the calling program. e.g., 3.12

Subroutine GET will bring an ascii character from TTI into ACO.

```
          .LOC  µ
START:      |
            ↓
LOOP:     JSR GET          ; get an ascii character in
            |
            ↓              } process and compute
          JMP LOOP
            |
            ↓
          .END
GET:      STA  3,GET1      ; same return address
          SKPDN  TTI
          JMP .-1
          DIAC 0,TTI
          JMP @ GET1       ; return
GET1:     0                ; storage location to save
                           ; return address
```

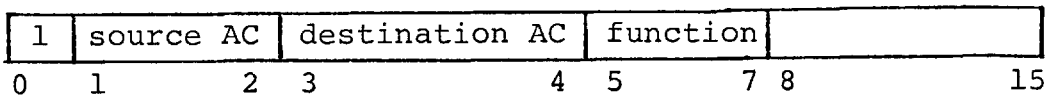
e.g., 3.13 Subroutines within subroutine



Section 3.4 Arithmetic and Logical Instructions (ALI)

The ALI's are used for performing specific arithmetic or logical operations between accumulators.

Instruction format



A 1 in bit 0 indicates an ALI.

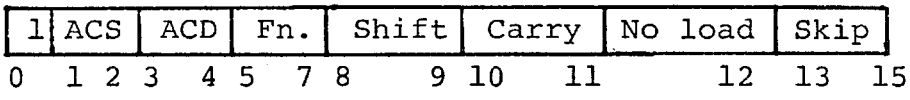
In the following text, the source AC is called ACS, and the destination AC is called ACD.

The arithmetic function is specified by $2^3 = 8$ possible operations:

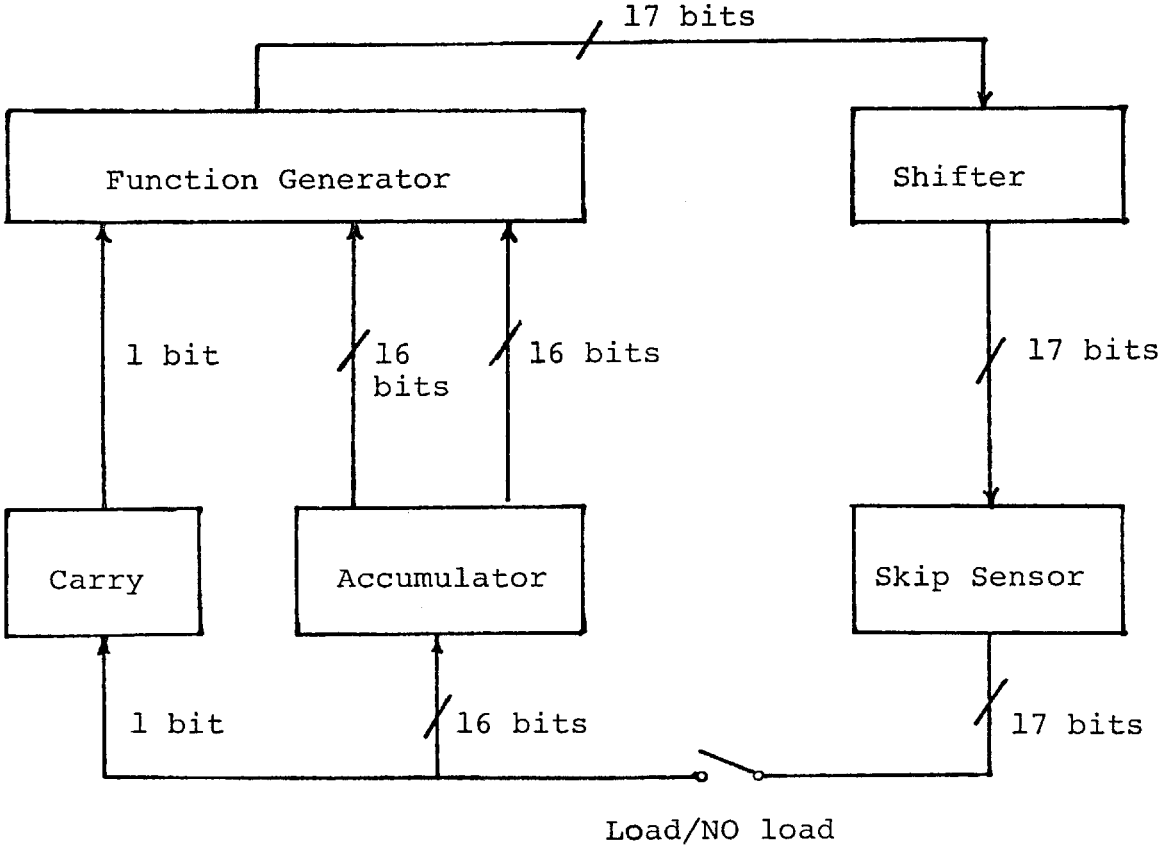
ALI	Function
COM ACS, ACD	the 1's complement of ACS is deposited into ACD (1's complement = all 1's changed to 0's and vice versa)
NEG ACS, ACD	the 2's complement of ACS is deposited into ACD (then (ACD) = -(ACS))
MOV ACS, ACD	copy (ACS) into ACD
INC ACS, ACD	deposit (ACS) + 1 into ACD
ADD ACS, ACD	deposit (ACS) + (ACD) into (ACD)
SUB ACS, ACD	deposit (ACD) - (ACS) into (ACD)
ADC ACS, ACD	deposit (ACD) + 1's complement of (ACS) into ACD
AND ACS, ACD	deposit (ACD) logical and (ACS) into ACD.

ACD= ACS is allowed. If ACS is not also ACD, then the original (ACS) is preserved.

Once the function has been performed, the result can be operated upon before it is loaded into ACD. These additional operations make up bits 8 through 15.



Diagrammatic representation of the arithmetic and logical unit



Signed and Unsigned operation*

Instructions in the Arithmetic and Logical class assume unsigned arithmetic and the prevalence of signed numbers must be taken into account by the programmer.

Carry Field

The value of the carry supplied to the function generator prior to performing the function is called the base value of the carry bit. This base value may be affected by the results of the function performed. If the function performed in the function generator results in an overflow, then the base value of the carry bit is complemented.

The following conditions will cause overflow:

Function	Unsigned Conditions Causing Overflow
ADD ACS, ACD	(ACS) + (ACD) $> 2^{16} - 1$
SUB ACS, ACD	(ACS) $\leq$ (ACD)
NEG ACS, ACD	(ACS) = 0
INC ACS, ACD	(ACS) = $2^{16} - 1$
ADC ACS, ACD	(ACS) < (ACD)

The initial or base value of the carry bit is specified in the instruction by bits 10 and 11. (The mnemonic is appended to the 3-letter function mnemonic).

Carry field	If the function is appended with	Base value of Carry is
00	- (nothing)	the current state of carry
01	Z	0
10	O	1
11	C	the complement of the current state of Carry

Examples 3.14

SUB 0,0 clears AC0 and complement carry

SUBO 0,0 clears AC0 and carry

Shift Field

After a function has been performed, the result may be rotated left or right as specified by bit 8 and 9 of the instruction. (The mnemonic is appended to the 3 or 4 letter function and carry mnemonic, after the carry mnemonic, of one occurs).

Shift field	Mnemonic	The result is
00	-	unchanged.
01	L	rotated left by 1 bit: bit 15 → bit 14, 14 → 13, ..., 1 → 0, 0 → Carry, Carry → 15.
10	R	rotated right by 1 bit: bit 1 → 2, 2 → 3, ..., 15 → Carry, Carry → 0.
11	S	bytes* are swapped: bit 0 through 7 are swapped with bit 8 through 15. Carry is unchanged.

It can be seen that shifting to the left or right is essentially a circular rotation with the carry bit.

Example 3.15

Write an instruction which multiplies the contents of AC0 by 2 and puts the result into AC2.

```
MOVZL  0, 2     ; Z sets carry = 0.
                ; Then L rotates everything left
                ; 1 bit and puts the 0 carry
                ; into bit 15.  The answer is
                ; then put into AC2.
```

Skip Field

After the function has been performed and the results shifted, the results may be tested to see whether or not the next instruction may be skipped. The conditions are based on the specifications of bits 13, 14 and 15 in the instruction. (The following mnemonics can be used after ACD).

bits 13, 14 & 15	The mnemonic is	The shifted result will be tested and the program will
000	-	never skip
001	SKP	always skip
010	SZC	skip if carry bit is zero
011	SNC	skip if carry bit is nonzero
100	SZR	skip if the result (bits 0 through 15) is zero
101	SNR	skip if the result is nonzero
110	SEZ	skip if either the carry or the result or both are zero
111	SBN	skip if both the carry and the result are nonzero

Load/No Load Field

Once the function has been performed, the shifting completed, and the decision for skip made, the result may or may not be loaded into ACD depending on bit 12 of the instruction.

Mnemonic	bit 12	Result .3
	0	loaded into ACD
#	1	not loaded into ACD, leaving ACS and ACD unchanged

The mnemonic is appended to the end of the complete function mnemonic.

Example 3.16

Write a routine to test the sign of a signed number in AC1 without destroying it.

```
        MOVL#  1, 1, SNC        ; sign bit moved to Carry bit
                                ; The Carry bit is tested
        JMP POS                 ; Carry = 0, go to routine
                                ; labelled POS
        --------                ; Carry = 1, implies the number
        --------                ; is negative
POS:    --------
        --------
        --------
```

3.4.1 Conditional Branching and Looping

The ALI's can be used to implement the IF-THEN-ELSE and REPEAT UNTIL structures.

IF-THEN-ELSE

Example 3.16 is an illustration of using ALI's in implementing the IF-THEN-ELSE structure.

Example 3.17

Write a routine to test whether AC0 is zero.

```
TEST:    MOV# 0, 0, SNR      ; skip on nonzero
                             ; result
         JMP ZERO            ; if [ACO] = 0, go to routine
                             ; ZERO
         --------            ; if [ACO] ≠ 0, proceed from here
ZERO:    --------
         --------
         --------
```

REPEAT UNTIL

Example 3.18

Write a program to output a character string to the TTO. Termination of the program is made upon detecting a null character.

```
;       THIS PROGRAM WILL OUTPUT A CHARACTER STRING TO THE TTO.
;TERMINATION IS DONE UPON DETECTING A NULL CHARACTER
;
        .LOC 40
;
;SUB. TYPE WILL OUTPUT A CHAR TO THE TTO
;
TYPE:   STA 3,SAVE              ;SAVE RETURN ADDS
        SKPBZ TTO               ;TEST IF BUSY=0
        JMP .-1
        DOAS 0,TTO              ;OUTPUT THE CHARACTER
        JMP @SAVE               ;RETURN TO THE CALLING
                                ;PROGRAM
SAVE:   0
;
        .LOC 1000
START:  LDA 0,POINT             ;INITIALIZE  WORD
        STA 0,WORD
LOOP:   LDA 0,@WORD             ;GET A CHAR.
        MOV 0,0,SNR             ;TEST FOR NULL CHAR.
        HALT                    ;YES,STOP
        JSR TYPE                ;GO TO SUBROUTINE TYPE
        ISZ WORD                ;GET ANOTHER CHAR
        JMP LOOP
;
POINT:  .+2                     ;POINT POINTS TO THE STARTING
                                ;ADDS OF STRING
WORD:   0                       ;RESERVED FOR CHAR POINTER
        117                     ;ASCII OF O
        125                     ;         U
        124                     ;         T
        120                     ;         P
        125                     ;         U
        124                     ;         T
        000000                  ;NULL CHAR
        .END
```

Notes: (i) .+2 means current location plus two.

(ii) Subroutine TYPE is located in page zero so that it can be reached by any call anywhere in core.

3.4.2 Programming Tricks

1. Generate the indicated constants.

```
SUBZL  AC, AC    ; generate +1
ADC    AC, AC    ; generate -1
ADCZL  AC, AC    ; generate -2
```

2. Subtracting 1 from an accumulator without using a constant from memory.

```
NEG  AC, AC
COM  AC, AC
```

3. Test an accumulator for -1

```
COM#   AC, AC, SZR
JMP    ---           ; not -1
---    ---           ; -1
```

4. Test if two accumulators are equal

```
SUB#  ACS, ACD, SNR
JMP   ---            ; equal
---   ---            ; not equal
```

5. Unsigned magnitude compare

```
SUBZ#  ACS, ACD, SNC     ; skip if [ACD] ≥ [ACS]
ADCZ#  ACS, ACD, SNC     ; skip if [ACD] > [ACS]
```

3.4.3 More Examples

Example 3.19

Write a routine to perform exclusive -OR.

[AC1] = operand A

[AC0] = operand B

[AC1] = result of A ∀ B

Algorithm used is

A∀B = A + B - 2(A∧B)

```
EXOR:  MOV   1,2    ; put [AC1] into AC2
       ANDZL 0,2    ; 2(A∧B) into AC2
       ADD   0,1    ; [AC1] = A + B
       SUB   2,1    ; A∀B = A + B - 2(A∧B)
```

Example 3.20

Write a routine to perform double precision addition.

[AC1] = high order word of operand A

[AC0] = low order word of operand A.

[AC3] = high order word of operand B.

[AC2] = low order word of operand B.

```
DADD:  ADDZ  0, 2  SZC   ; to check for overflow when
                         ; adding 2 low order words
       INC   3, 3        ; Carry = 1, add it to
                         ; one of the higher order
                         ; words
       ADD   1, 3        ; add the higher order
                         ; words
```

[AC3] = high order word of the result A + B.

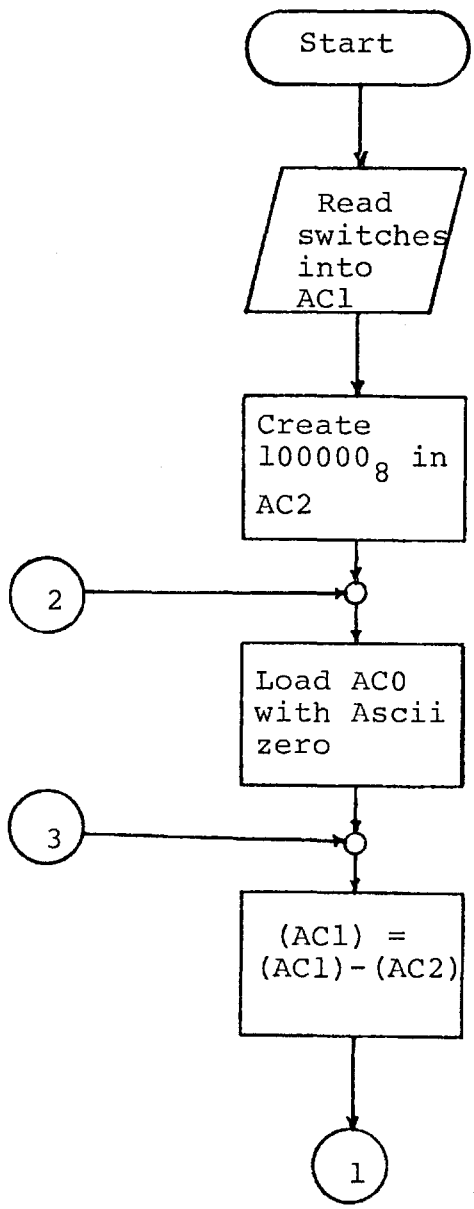
[AC2] = low order word of the result A + B.

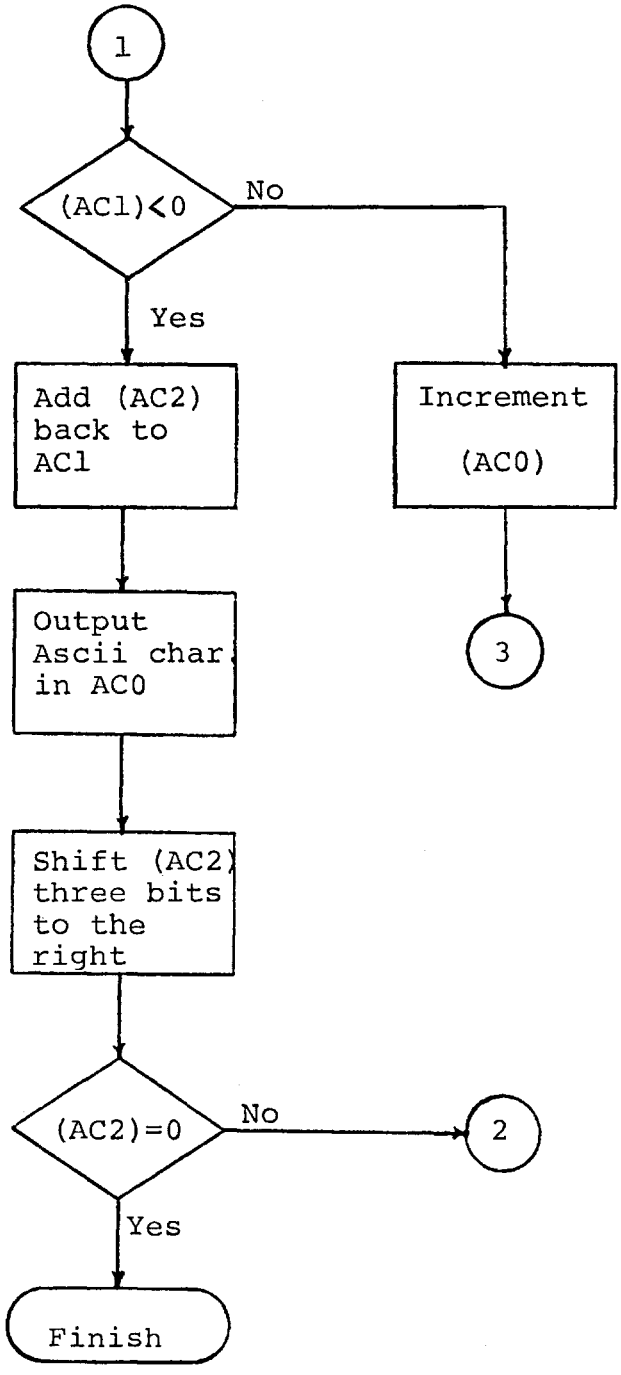
Suggested exercise

Appendix A, No. 6, 7 & 8.

Section 3.5 Example 3.21

Write a program to convert a binary number in the accumulator into its octal equivalent and output the result to the TTO. The binary number should be read in from the switches.




1
(AC1)<0
No
Yes
Add (AC2) back to AC1
Increment (AC0)
Output Ascii char in AC0
3
Shift (AC2) three bits to the right
(AC2)=0
No
2
Yes
Finish

```
;       THIS PROGRAM WILL CONVERT A BINARY NUMBER IN THE AC
;INTO ITS OCTAL EQUIVALENT AND THE RESULT IS OUTP'IT
;ON THE TTO. THE BINARY NO. IS READ FROM SWITCHES.
;
        .LOC 40
;                               OUTPUT SUBROUTINE
OUT:    STA 3,SAVE
        SKPBZ TTO
        JMP .-1
        DOAS 0,TTO
        JMP @SAVE
SAVE:   0
;
        .LOC 100
START:  HALT                    ;SET SWITCHES
        READS 1
        SUBZR 2,2               ;CREATE 100000 OCTAL IN AC2
LOOP:   LDA 0,C60               ;(AC0)=ASCII ZERO
        SUBO 2,1,SNC            ;STILL POSITIVE IF CARRY=0
        INC 0,0,SKP             ;INCREMENT ASCII CHAR
        ADD 2,1,SKP             ;CARRY=1. RESULT IS NEGATIVE.
                                ;ADD (AC2) BACK TO (AC1)
        JMP .-3
        JSR OUT                 ;OUTPUT CHAR
        MOVZR 2,2               ;SHIFT (AC2) 1 BIT RIGHT
        MOVZR 2,2
        MOVZR 2,2,SZR           ;TEST FOR LAST DIGIT
        JMP LOOP                ;NO,CONTINUE
        JMP START               ;YES, GET ANOTHER CHAR
;
C60:    60                      ;ASCII ZERO
        .END
```

Suggested Exercise

Appendix A, No. 9.

Chapter 4 Testing and Debugging[3, 4]

Errors will usually fall into three categories:

1. Syntax Error - the language was used improperly.
2. Logical Error - the program has been assembled, but it is not running.
3. Algorithmic Error - the program is running, yet the result is incorrect.

4.1 Debugging techniques

1. Syntax Error - The assembler will usually give some indications during assembly as to where the errors occur and what types they are. The programmer has to correct those mistakes indicated by the assembler.
2. Logical Error - The following techniques are helpful in detecting and correcting logical errors.
   (a) By using the switches on the front panel of the NOVA computer the program can be checked step by step.
   (b) By using HALT instructions, a running program can be stopped at appropriate places to allow examination of intermediate results.
   (c) To trace the flow of a program, extra output statements can be included in different places of the program. From the characteristic indications given by these output statements, one should be able to locate approximately where the error occurs.

3. Algorithmic error - After all the bugs have been removed, and if the program still does not give any valid answers to the problem, then there must be something wrong with the logic used. The programmer then has to re-evaluate the algorithms utilized in the formulation of the program and sub-programs.

4.2 Prevention of Bugs

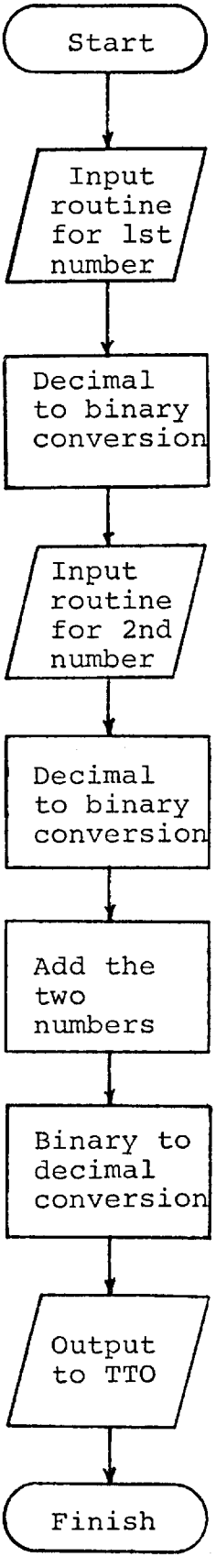
1. Structured Programming

A structured program is less prone to errors. Debugging is made less difficult because of the simplicity in control structures associated with structured programs.

Modular programming is an important technique when one starts to write long assembly language programs. Very often "bugs" can be singled out in a particular module, and, therefore, debugging is confined to that module only. In assembly language, subroutines can be used to implement modular programming.

Example 4.1

Write a program which will add two numbers together. Input is from the TTI keyboard and the result is output to the TTO.



Flowchart solution to add two decimal numbers

When the solution is implemented into a program, it will look somewhat like the following.

```
          .LOC 40
;         STORAGE LOCATIONS AND CONSTANTS

          -----
          -----
          -----

          .LOC 100
;PROGRAM TO ADD TWO NUMBERS
;INPUT: FROM TTI KEYBOARD. OUTPUT: TTO

START:    JSR INPUT          ;INPUT A NUMBER
          JSR DTOB           ;DECIMAL TO BINARY CONVERSION
          JSR INPUT          ;GET SECOND NUMBER
          JSR DTOB
          JSR ADDIT          ;ADD TWO NUMBERS
          JSR BTOD           ;BINARY TO DECIMAL CONVERSION
          JSR OUTPT          ;OUTPUT RESULT
          JMP START          ;READY FOR ANOTHER CALCULATION

;SUBROUTINES

INPUT:    -----              ;INPUT ROUTINE FROM TTI
          -----
OUTPT:    -----              ;OUTPUT ROUTINE TO TTO
          -----
DTOB:     -----              ;DECIMAL TO BINARY ROUTINE
          -----
BTOD:     -----              ;BINARY TO DECIMAL ROUTINE
          -----
ADDIT:    -----              ;ADDITION ROUTINE
          -----

          .END
```

There are five modules in the program, each of which can be fully tested and debugged before linking together. Attention has to be paid when interfacing one module to another.

2. Liberal use of comment

A well-commented program will ease the effort of understanding that program at a later date. However, comment statements should be placed neatly in the program to minimize confusion with the main program.

Example 4.2

The following is form is desirable.

```
; a description of what is being done in the routine
; input and output conditions
; ----
; ----
              .LOC n
  START:      ------      ; It is good to have
              ------      ; all comments aligned
              ------      ;
```

While this arrangement should be avoided.

```
       ----               ; ----
       ----               ; (comment) --- not enough space
; continue   this will make the
          }
; continue   instruction statements less prominent.
```

Note: Comments that add no information or that are redundant should be avoided.

E.g., ADD 0, 1 ; Add accumulator 0 and 1.

Chapter 5 Interrupt Programming

When a program has gained control of the CPU, usually there is no way of interrupting the execution of the program by other resources (e.g., peripherals, other programs) within the system. Until the current program terminates itself, no other resources are allowed to use the CPU. This is not desirable since situations often arise where some more urgent jobs are waiting to be serviced or the operater wishes to communicate with the processor. Moreover, slow devices similar to the teletype and line printer can waste a lot of valuable execution time since the CPU has to wait on these devices.

The introduction of the interrupt programming is aimed at solving these problems. Namely,

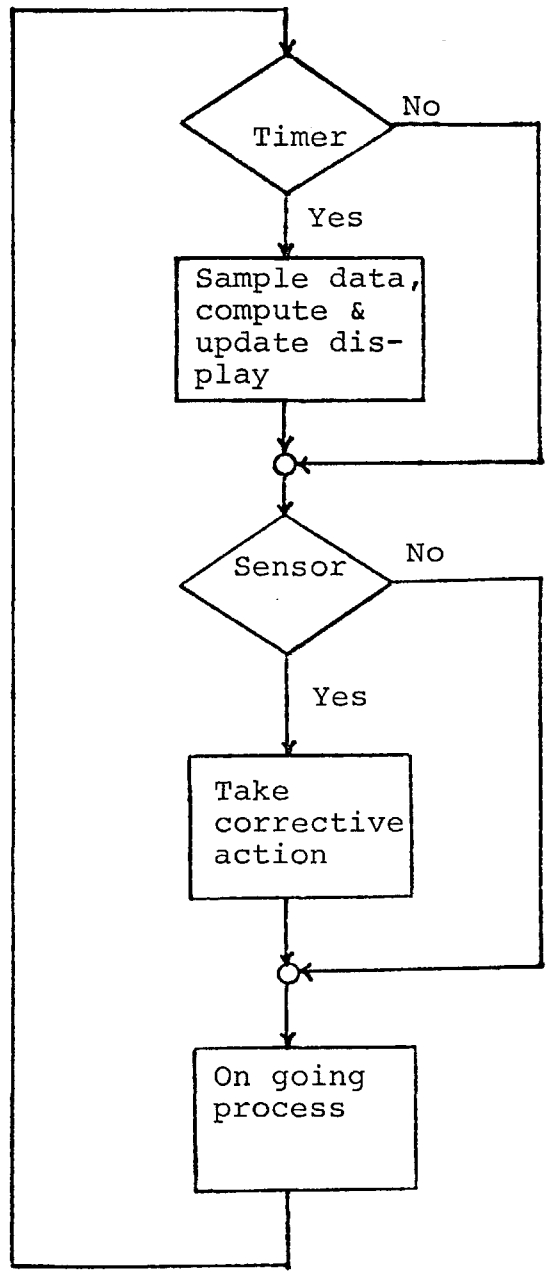
1. It allows an external device to temporarily suspend the main program in execution. Several external devices can interrupt the system at the same time. In this case, access to the system will be given to the device which has the highest priority*.
2. It improves overall system efficiency by utilizing time normally wasted in waiting loops encountered while servicing slow devices.

Example 5.1

Consider the situation where a number of external parameters are sampled by the computer once every second and some mathematical computation is to be performed. Also present is an On/OFF sensor which monitors an extremely

critical condition of the system.

Using non-interrupt programming, the program flow may appear as follow.

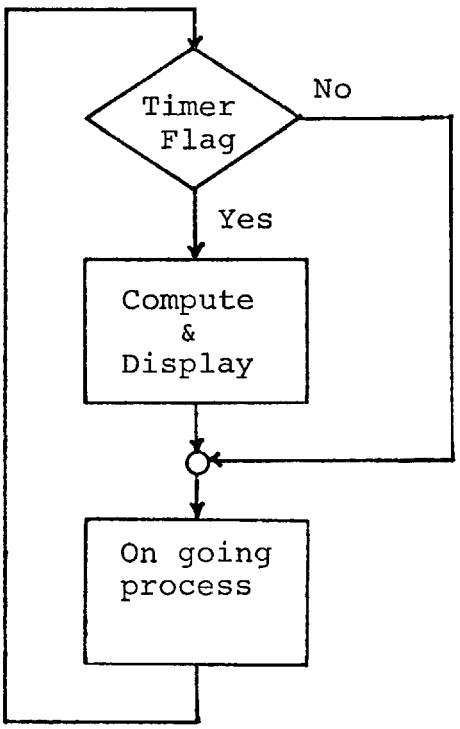


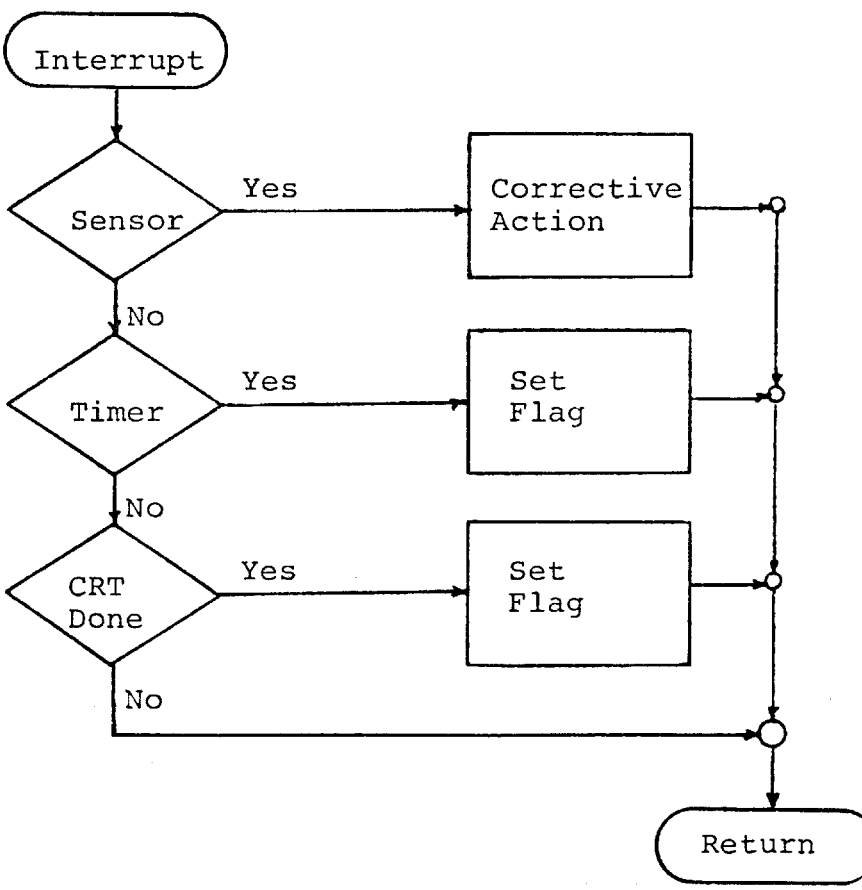
It is assumed that on going process is done before next timing signal comes in.

In this program, checking the state of the sensor is part of the program which may be time consuming. In addition to this, there will be no way to detect the system status outside the checking procedure. A situation may appear where the alarm is turned on just after a check and because of the rigidity of the program flow, the corrective action will not be in effect until the next check. The price for slow reaction may be disastrous.

A way to solve this problem is to give the sensor the capability to interrupt the normal flow of the program. The computer must respond to the interruption by initiating the corrective action immediately. The main flow of the program will resume after the corrective action has been performed.

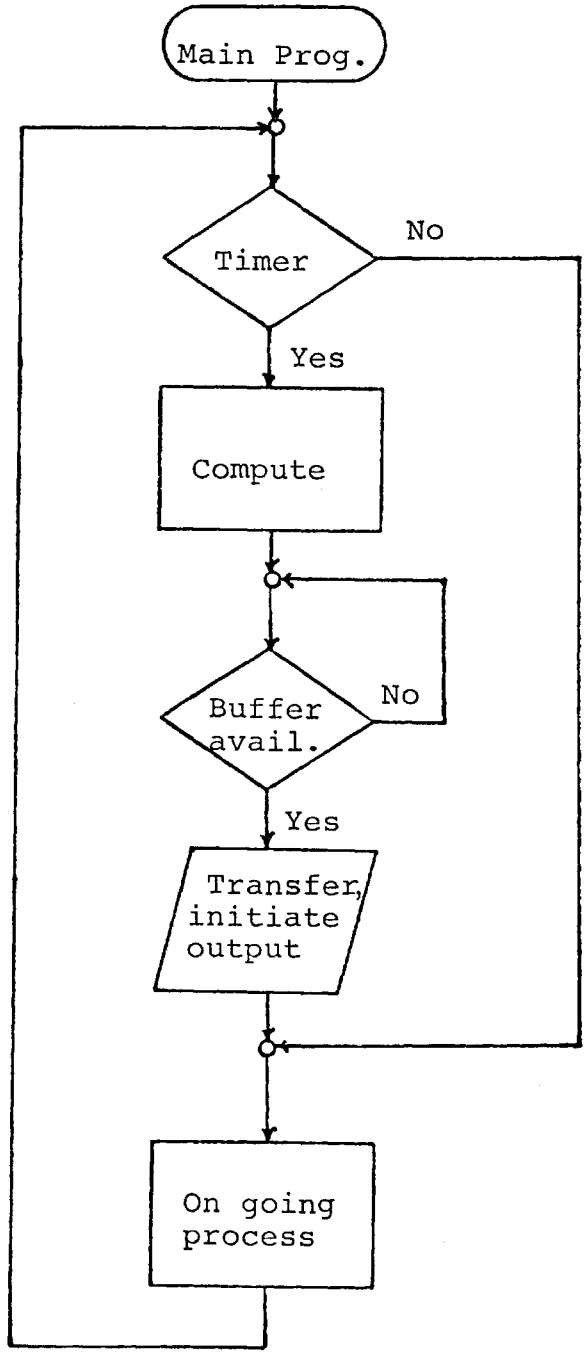
With the interrupt facility implemented, the program may appear as follow:





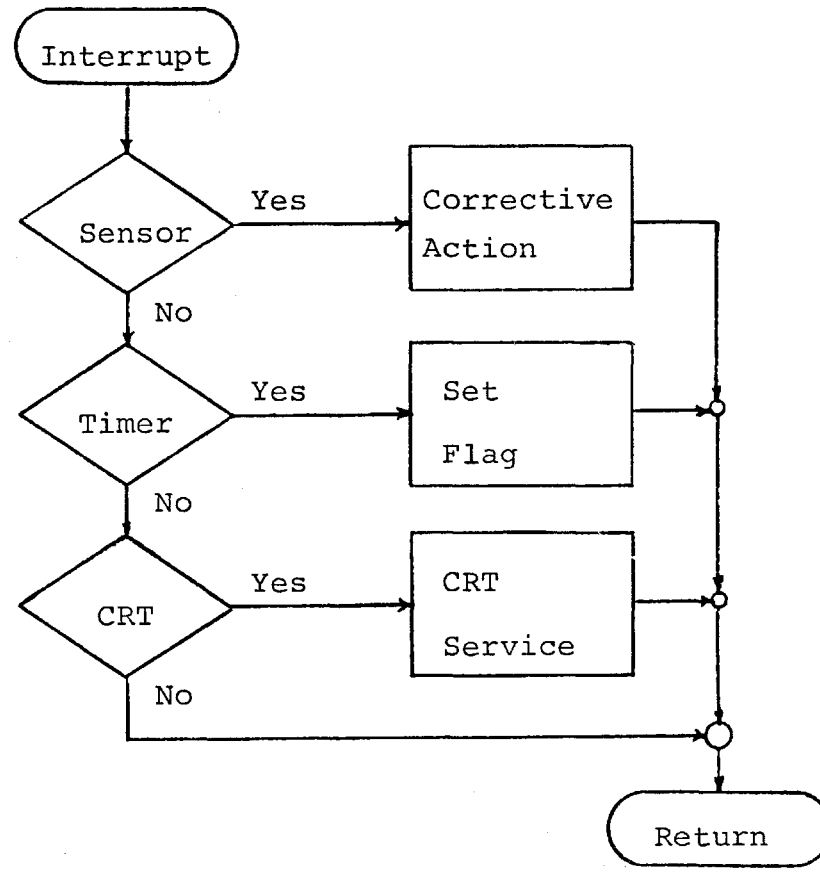
In this interrupt system, priorities have been assigned to the three devices in this order: sensor, timer and display. This is done by checking the sensor first, the timer next and the display unit last in the interrupt routine. Basically all peripheral devices are sharing one interrupt line. Priority assignment is usually done by softward, although hardware priority schemes can also be implemented.

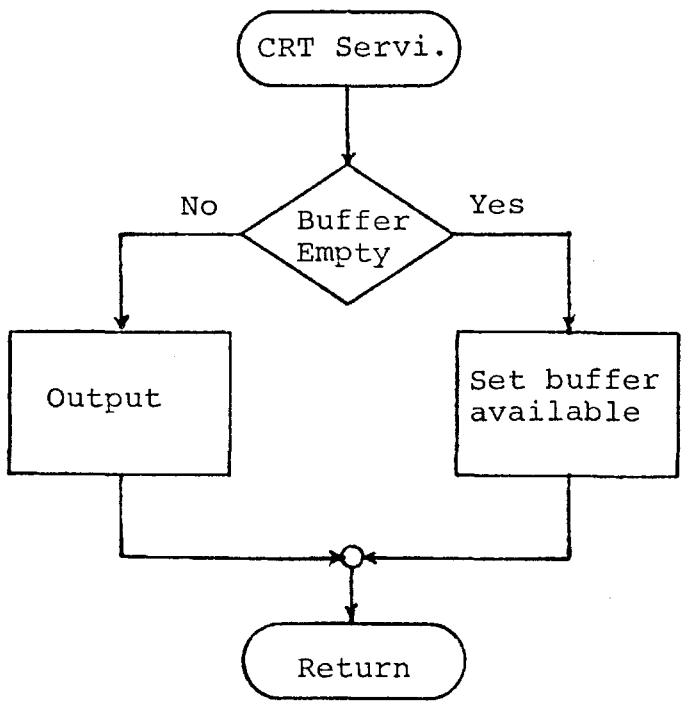
The second problem dealing with slow devices (and also fast mass storage devices) is generally solved using buffered I/O techniques whereby a block of information is assembled before transferring between the device and the main program.



Main Program

Interrupt routine





CRT Service Routine

Remarks

A very general approach has been taken to introduce the concept of interrupt programming. Interrupt programming techniques associated with the Nova is depicted in Ref. 2 & 5.

Basically the two most widely used methods in process synchronization have been explored. Namely, they are

1. Flag checking system.
2. Interrupt system.

There are advantages and disadvantages associated with each method. In most systems, both the two methods are implemented to make the system more efficient.

Appendix A

1. Write a program to move a block of information from one place of core to another using base register addressing.
2. Write the same program using autoincrementing locations.
3. Write a program which will accept characters from the TTI and echo them back to the TTO immediately at the same time.
4. Write a program which will store a string of characters entered via the TTI and output the string to TTO when a special key is struck.
5. Write a program which will convert the incoming ascii digits from the TTI into their true binary values.
6. Write routines for the following:
   (a) Inclusive OR
   (b) Double precision subtraction.
   (c) Test for odd and even numbers.
   (d) Absolute value.
7. Write a program to accept a series of octal Ascii digits terminated by a non-digit characters, transform the series into a positive binary integer and store this number in memory. The number should be echoed back to the TTO.
8. Write a program to zero all memory locations.
9. Solve the binary to octal conversion problem with another method.

Appendix B Glossary[4]

The terms to be explained in this glossary are in the order of their appearance in the manual.

Chapter 1

Computer program - Computer language: In order to solve a computation problem, its solution must be specified in terms of a sequence of computational steps, each of which may be effectively performed by a human agent or by a digital computer. Systematic notations for the specification of such sequences of computational steps are referred as computer (programming) languages. A specification of the sequence of computational steps in a particular programming language is referred to as a computer program.

Debugging - Errors within a program are usually referred as "bugs". The process of removing bugs from a program is called "debugging".

Algorithm - An algorithm is a clerical procedure which can be applied to any of a certain class of symbolic inputs and which will eventually yield, for each such input, a corresponding symbolic output.

Flowchart - Flowcharts are pictorial representations of algorithms providing the programmer with an indication of the flow of the program. It is a combination of symbols with English and

mathematical statements that clearly defines every step in solving the problem. There are a number of symbols, each of which indicates the type of operation to be performed. More detailed explanations of the operation are contained within the symbol.

Structured Programming - It is concerned with improving the programming process through better organization of programs and better programming notation to facilitate correct and clear descriptions of data and control structures. Usually a structured program is easier to understand, modify and debug.

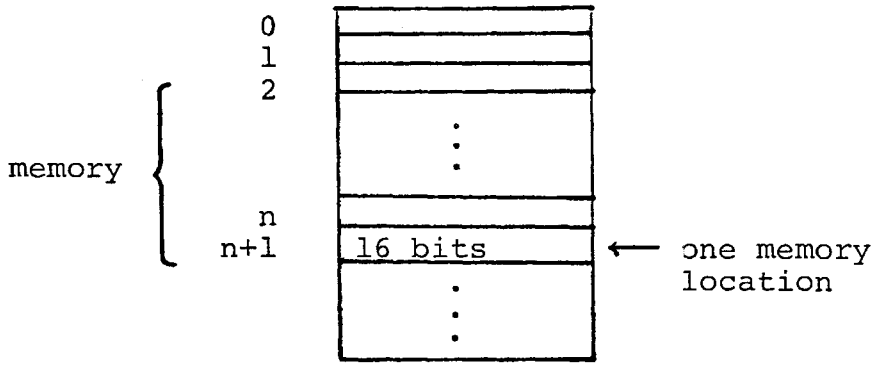
Branching - A transfer of control from one place to another within a program.

Modular programming - A program module can be defined as a logically self-contained and discrete part of a larger program and which performs a specific function. A properly constructed module has only one entry point and only one exit point. The purpose of it is to break a complex task into smaller and simpler subtasks which facilitates writing correct programs.

Chapter 2

Memory
Memory location - Memory is one of the more important parts of a computer system. It is used to store information which includes programming instructions and data. In Nova, programming instructions and data are in a 16-bit word format, the memory space capable of storing

one piece of 16-bit information (word) is called a memory location.



Accumulator - For the Nova, an accumulator is a 16-bit register used for temporary storage during data processing.

Binary Codes - Codes that use only two symbols 0 and 1 are called binary codes.

Machine language - Machine language is a primitive programming language which is immediately executable by the machine concerned. It is usually coded with binary.

Octal number system - a number system that uses 8 as its base.

Assembly language - it is a language in which all operators and all operands are normally represented by names chosen for their explanatory and mnemonic power. It can be viewed as a mnemonic equivalence of the machine language.

Mnemonic - It refers to the name given to a machine language coding to facilitate memorizing the code and its function.

Pseudo-operation - pseudo-operation is not an instruction to be performed, but directive to the assembler.

Assembler - An assembler is a program that facilitates the preparation of programs at the assembly language level by taking mnemonics of individual instruction or data and converting them into the corresponding binary coding. It is usually provided by the manufacturer of the computer.

Auto-incrementing - An auto-incrementing (decrementing)
Auto-decrementing location has the characteristic that its content will be incremented (decremented) by 1 when being addressed in a specific way. (Indirect addressing in case of Nova).

Indirect Addressing - If an instruction references data indirectly by pointing to the address of the data rather than the data itself, then the technique used is called indirect addressing.

Interrupt programming - Interrupt programming allows an external device or a more urgent job to interrupt the current activity of the CPU and take over it. The execution of the current program will be resumed after the interrupting program has finished, providing there is no other higher priority request

waiting to be serviced.

delimiter - A delimiter is an item of lexical information whose form and/or position in a source program denotes the boundary between adjacent syntactic components of that program.

Chapter 3

Memory Reference Instructions (MRI) - Instructions that reference, modify the content of or transfer control to, a memory location are called memory reference instructions.

Program counter (PC) - A PC is a counter which keeps track of the flow of the program by pointing each time to the instruction to be executed. After the current instruction has been finished, PC will point to the next instruction.

Displacement (D) - In Nova, displacement refers to the value of bits 8 through 15 of an MRI. For page zero addressing, $0 \leq D \leq 377_8$

and for the other modes of direct addressing

$-200_8 \leq D \leq 377_8$

Direct addressing - If an instruction references data directly by pointing to the data, then the technique is called direct addressing.

Page Zero - In Nova terminology, location 0 through location 377_8 is called page zero.

2's complement of a number - the 2's complement of a number X is equal to the largest possible number (for the Nova, largest possible number is 177777_8) plus 1 minus X.

Peripheral equipment - all the devices that are attached to the computer and serve either as the means of feeding raw data or file data into the system or receiving results or updated files from the system are referred as peripheral equipments.

Central Processing Unit (CPU) - It is used to describe elements that carry out a variety of essential data manipulations and controlling tasks at the heart of the computer.

Flip-flop - It is a bistable logic unit generally used as a basic storage element in various registers inside the computer.

Buffer - A buffer is an area of storage which temporarily holds data that will be subsequently delivered to a processor or input/output transducer.

Teletype - It is a peripheral equipment which works like a typewriter.

The American Standard Code for Information Interchange (ASCII) It is a seven-bit code used to denote different characters.

Paper tape - It is a peripheral equipment used as a storage medium for the preparation, storage and transmission of data in various appli ations.

Parity check - It is an extensively used error-checking facility provided to ensure correct recording of data, its input into the computer system, and its transfer within the system.

Subroutine - A subroutine is a portion of a program which is logically independent and performs a specific task necessary for the execution of the program.

Signed and Unsigned number - For the Nova, a number can be regarded as signed by the programmer. Bit 0 = 0 denotes positive while bit 0 = 1 represents negative. The Nova will always presume unsigned operations.

Carry - it is an extra bit attached to the left of bit 0 during arithmetic and logical operations and is used to test for sign and overflow condition.

Chapter 5

Priority - It is a method of scheduling to determine which resource in the system can have privileged access of the CPU than the other resources.

REFERENCES

1. Data General Corporation, "Fundamentals of Mini-Computer Programming", 1973.
2. Physics Department, McMaster University, "Introduction to the Data General Nova", Physics 4D6 course manual.
3. W.W. Peterson, "Introduction to Programming Languages". Prentice Hill, 1974.
4. A Ralston & C.L. Meals, "Encyclopedia of Computer Science", Petrocelli/Charter, 1976.
5. Data General Corporation, "Introduction to Programming the Nova Computers", 1972.