UIUCDSC-R-71-476

ILLIAC III COMPUTER SYSTEM MANUAL:
TAXICRINIC PROCESSOR

VOLUME 3

By

Bernard J. Nordmann, Jr.

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

August 24, 1971

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

This work was supported by Contract AT(11-1)-1018 with the U. S. Atomic Energy Commission through September 30, 1970. Current support is under Contract AT(11-1)-2118 with the above agency.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

ACKNOWLEDGEMENT

The conceptual design of the Taxicrinic Processor of the Illiac III Computer System is largely the work of three individuals: Roger E. Wiegel, Bruce H. McCormick and the author. Dr. R. M. Lansford emphasized the importance of the associative addressing scheme wherein the processor can automatically enter the Segment Name Table for additional base descriptors, as needed. This facility, central to an effective Operating System, has been incorporated into the design of the Taxicrinic Processor.

Familiarity with the Illiac III Reference Manual, Volumes 1, 2, and 4, edited by B. H. McCormick and B. J. Nordmann, Jr., 1971, is assumed.

Bruce H. McCormick and John P. O'Donnell assisted in the editing of this volume of the TP Manual. Thanks are also due to Mrs. Roberta André and Mrs. Judy Arter for their labor on the many revisions of this volume and to Stan Zundo and the countless other members of the drafting department who labored over the drawings and flow charts. Finally, a special thanks goes to Dennis Reed and his magical ITEK machine for the laborious task of the reduction and printing of the flow charts without which this manual would have been impossible.

5. INSTRUCTION SEQUENCES

5.1 TP-Executed Primitive Instructions

5.1.1 Operand Stack Instruction Sequences

- 5.1.1.1 OS Instruction Sequence Descriptions
- 5.1.1.2 POP, STORE, PUSH and LOAD Control Logic
- 5.1.1.3 DUP and SLUFF Control Logic
- 5.1.1.4 XCH Control Logic

5.1.2 Multiple Cycle Instruction Sequences

- 5.1.2.1 Initial & Final Multi-Cycle Sequences
- 5.1.2.2 Main Multi-Cycle Sequence
- 5.1.2.3 Multi-Cycle Operation Sequence
- 5.1.2.4 Edit Sequence

5.1.3 Other Data Transfer Instruction Sequences

- 5.1.3.1 Assign Sequence Description

5.1.4 List Processing Instruction Sequences

- 5.1.4.1 List Processing Sequence Descriptions
- 5.1.4.2 SL and SR Control Logic
- 5.1.4.3 GETC and PUTC Control Logic
- 5.1.4.4 INCL and INCR Control Logic
- 5.1.4.5 DECL and DECR Control Logic

5.1.5 Unary Operation Instruction Sequences

5.1.5.1 Logical Unary Instructions

- 5.1.5.1.1 Logical Unary Sequence Descriptions
- 5.1.5.1.2 ONE, ZERO and NOT Control Logic
- 5.1.5.1.3 COUNT and BIT Control Logic

5.1.5.2 Algebraic Unary Instructions

- 5.1.5.2.1 Algebraic Unary Sequence Descriptions
- 5.1.5.2.2 ABS, NEG and MNS Control Logic
- 5.1.5.2.3 TA Control Logic
- 5.1.5.2.4 CVL Control Logic

5.1.6 Boolean/Shift Instruction Sequences

5.1.6.1 Boolean Operation Instructions

- 5.1.6.1.1 Boolean Operation Sequence Descriptions

- 5.1.6.1.2 AND, OR, XOR and EQV Control Logic
 - 5.1.6.2 Boolean Transfer Instruction
 - 5.1.6.2.1 Boolean Transfer Sequence Descriptions
 - 5.1.6.2.2 SET and RESET Control Logic
 - 5.1.6.3 Shift Instructions
 - 5.1.6.3.1 Shift Sequence Description
 - 5.1.6.3.2 LS and RS Control Logic
- 5.1.7 Algebraic/Logical Compare Instruction Sequences
 - 5.1.7.1 Basic Compare Sequence Description
 - 5.1.7.1.1 A/L Compare Sequence
 - 5.1.7.1.2 A/L Compare Control Logic
 - 5.1.7.2 A/L Compare Instructions
 - 5.1.7.2.1 A/L Compare Sequence Descriptions
 - 5.1.7.2.2 CPRA and CPRL Control Logic
 - 5.1.7.2.3 TEST and TESTM Control Logic
- 5.1.8 TP Utility Instruction Sequence
 - 5.1.8.1 Specify Instruction
 - 5.1.8.1.1 Specify Sequence Description
 - 5.1.8.1.2 Specify Control Logic
 - 5.1.8.2 Location Instruction
 - 5.1.8.2.1 Location Sequence Description
 - 5.1.8.2.2 LOC Control Logic
- 5.2 AU-Executed Primitive Instructions
 - 5.2.1 Basic AU Sequences
 - 5.2.1.1 Basic AU Sequence Descriptions
 - 5.2.1.2 AU ENTRY Control Logic
 - 5.2.1.3 AU RESULT Control Logic
 - 5.2.2 Arithmetic Instruction Sequences
 - 5.2.2.1 Arithmetic Instruction Sequence Descriptions
 - 5.2.2.2 ADD and SUB Control Logic
 - 5.2.2.3 MPY and DIV Control Logic
 - 5.2.2.4 POLY Control Logic

5.3 PAU - Executed Primitive Instructions

5.3.1 PAU Instruction Formats

5.3.2 PAU Main Sequence

5.3.2.1 PAU Main Sequence Description

5.3.2.2 PAU Main Control Logic

5.3.3 PAU Utility Sequences

5.3.3.1 PAU Utility Sequence Descriptions

5.3.3.2 PAU ACCESS Control Logic

5.3.3.3 PAU FIELD ACCESS Control Logic

5.3.4 PAU FINI Sequence

5.3.4.1 PAU FINI Sequence Description

5.3.4.2 PAU FINI Control Logic

5.4 IU- Transmitted Primitive Instructions

5.5 Imprimitive Instructions

5.5.1 Imprimitive Instruction Descriptions

5.5.1.1 GOTO

5.5.1.2 CALL

5.5.1.3 EXECUTE

5.5.1.4 EXIT

5.5.2 Pointer Stack Manipulation

5.5.2.1 GOTO Stack Manipulation

5.5.2.2 CALL And EXECUTE Stack Manipulation

5.5.3 Pointer Name Permutation

5.5.3.1 Name Permutation Hardware

5.5.3.2 Name Permutation Sequence Description

5.5.3.3 Name Permutation Control Logic

5.5.4 Imprimitive Sequence

5.5.4.1 Imprimitive Sequence Description

5.5.4.2 Imprimitive Sequence Interrupt Description

5.5.4.3 Imprimitive Control Logic

5.6 System Instructions

- 5.6.1 Supervisor Call
- 5.6.2 Supervisor Return
- 5.6.3 Rename
- 5.6.4 Sleep
- 5.6.5 Activate TP
- 5.6.6 Increment and Check
- 5.6.7 OK OFF
- 5.6.8 Link
- 5.6.9 Who
- 5.6.10 Set Interrupt Mask
- 5.6.11 Reset Interrupt Mask
- 5.6.12 Reserve Unit
- 5.6.13 Load Task Register
- 5.6.14 Store Task Register
- 5.6.15 Interrupt Return

5.6 - Appendix: Miscellaneous Supervisor Problems

- 5.6.A.1 The Lockout Problem
- 5.6.A.2 The Interrupt Storage Problem
- 5.6.A.3 Interrupt Handling

5.1 TP - Executed Primitive Instructions

The primitive instructions in the Taxicrinic Processor correspond to the "machine language" instructions in most other computer systems. They are the most prevalent type of operation in the Illiac III computer.

As described in the programming manual, the primitive instructions consist of a single-byte mnemonic phrase followed by a 2 byte mask or count field or by 0 to 3 operand phrases. The operand phrases may be long (3 bytes) or short (1 byte), depending on the instruction. However, no group of operand phrases may ever use more than 4 bytes. The one exception to this is the SPECIFY instruction.

The reason for the 4 byte restriction is that in most of the primitive instructions, the various operands must be continuously available to the instruction sequence. Since the Instruction Register is only 4 bytes long, in order to avoid a lot of shifting back and forth, the operands may only use a total of 4 bytes. In SPECIFY each phrase is only used once. As it is scanned, the pre-and post operations are performed and after this is completed, the phrase does not need to be referenced again. Therefore, in this case, it is not necessary to restrict the total operand length.

In the primitive instructions all pre-operations specified by the operand phrases are performed before the actual instruction begins execution. The post-operations are performed when the execution is finished.

In the following sections, the primitive instructions are broken down into groups of instructions which are functionally related.

5. Instruction Sequences

This section of the TP Manual (Vol. III) describes the control sequences used to implement the instructions processed by the Taxicrinic Processors. These sequences, in turn, evoke the Basic Machine sequences described in Section 4 of the TP Manual (Vol. II).

Subsections of this volume are arranged according to the processor or unit which eventually executes the instruction: the Taxicrinic Processor, the Arithmetic Unit, the Pattern Articulation Unit, or the Interrupt Unit. A final section is devoted to the imprimitive instructions, which are treated somewhat differently from the primitive instructions.

Within each section the instructions are grouped according to their functional relationships. For each group there is a description of the instructions included within the group. This description reaches the level of gate transfers. Then there are separate descriptions of the detailed control logic used to implement each instruction. Where applicable flow charts are given for both levels of descriptions.

5.1.1 Operand Stack Instruction Sequences

5.1.1.1 OS Instruction Sequence Descriptions

This section describes the detailed control sequencing for the Operand Stack instructions. The instructions described here are those which exclusively use the Operand Stack. The flow charts are given at the end of this section. Those arithmetic and boolean operations which use the OS are described in their respective sections.

The POP and STORE sequences are exactly the same except that after the data has been read out of the OS, if the instruction is a STORE, the SCR is returned to its original position before the instruction began. This is done utilizing the STOR input for the OS READ sequence. Initially the OS ENTRY sequence is used to check for underflow and correct this if necessary. Then a memory access is begun to construct and bounds check the address where the data is to be stored (see Section 4.2 for the Memory Access sequencing). When the memory sequence returns to the Main Control sequence for the data to be stored, the OS READ sequence is activated. This loads the LR and DR with the desired data. Next a return is again made to the Memory Access sequence which completes the storage process and then returns. This completes the sequence.

The PUSH sequence stores data in the next available cell of the Operand Stack and then increments the SCR by the cell size of the data. Initially, the OS ENTRY sequence is used to check for overflow and correct this if necessary. Next a memory access is initiated to get the proper data from core. Finally this data is written into the Operand Stack using the OS WRITE Sequence.

The LOAD sequence is similar to the PUSH except at the beginning. Since a LOAD acts like a POP followed by a PUSH, the initial check by the ØS ENTRY sequence is for underflow. After this check, the SCR is decremented by the cell size, which effectively throws away the top cell of the Operand Stack. Now the PUSH sequence is followed except that since we just performed a POP, we know that there is room in the Operand Stack for another cell and we can therefore skip the overflow check and proceed directly to the memory access. From there onward the sequence is the same as the PUSH sequence.

The DUP sequence reads the top cell in the Operand Stack and then PUSH's it into the stack. This means, effectively that both a POP and a PUSH must be performed and that therefore the ØS ENTRY must check for both overflow and underflow. This is the reason for the DUP = 1 decision at the end of the underflow check in the ØS ENTRY sequence. By having the decision there, the necessity of going through the ØS ENTRY sequence twice is avoided. The rest of the DUP sequence is very simple consisting of an ØS READ sequence followed by an ØS WRITE sequence.

In the DUP sequence note that in the ØS READ sequence, the STØR = 1 option is used. This means (referring to the flowcharts of Section 4.5.1) that the SCR will be returned to its old value at the end of the read sequence and thus the value of the SCR is then correct when the ØS WRITE sequence begins. At the end of the write sequence the SCR points to the end of the newly duplicated stack entry.

The SLUFF sequence merely decreases the value of the SCR by the cell size, effectively throwing away the top ØS cell. However before this is done the sequence must use the ØS ENTRY sequence to check for underflow.

The XCH sequence is the most complicated ØS sequence. Its purpose is to exchange the two topmost cells in the Operand Stack. To do this it must "look backward" in the stack to make sure that the top two operands are in the TP fast storage. This is done by the ØS ENTRY sequence with CS2C set to "1".

Next, the next-to-top operand is read out of the Operand Stack by the ØS READ sequence and then stored in the AR (and SBR if the cell is a double word). During the read sequence STOR = 1, which means that at the completion of this sequence the SCR still points to the top of the stack.

After the next-to-top operand has been stored, the ØS READ sequence is used to read out the top operand in the Operand Stack. This time STOR = 0 so that the SCR points between the two operands at the end of this sequence.

Since this data is to be immediately written back into the Operand Stack in the next-to-top position, the SCR must be "moved" to a position in front of the next-to-top operand. Then the ØS WRITE sequence can be performed.

After this first write sequence is finished, the data that was originally stored in the AR and SBR is retrieved. This is then written into the top operand position of the Operand Stack, again using the ØS WRITE sequence. The two write sequences advance the SCR to the original SCR position. So at the end of the second ØS WRITE sequence the exchange is complete.

The interrupt cases are easily handled for these sequences. If an interrupt occurs in OS ENTRY, the logic in that sequence will undo as much as necessary to make the state of the stack legitimate. If the interrupt occurs in a memory sequence it will either be catastrophic or there will have been no permanent change in the data base. Since a successfully completed OS ENTRY will not change the data base, either, all of these sequences can be restarted at the beginning.

Operand Stack Instruction Sequences
POP and STORE Sequence Flow Charts

Operand Stack Instruction Sequences
PUSH and LOAD Sequence Flow Charts

Operand Stack Instruction Sequences
 DUP and SLUFF Sequence Flow Charts

5.1.1.2 POP, STORE, PUSH and LOAD Control Logic

The four instructions share the same control point OIT1, but conditionally activate four different tasks, namely, OIT1W, OIT1n, OIT1y and OIT1z. Each of those tasks in turn sets or resets the appropriate flip-flops for later use in the called internal sequences. A table of the states of the flip-flops after OIT1 is completed is shown in Figure 5.1.1.2/1.

On normal return from the MEMORY SEQUENCE DIRECT ENTRY in OIT4, instructions will go to two different paths, according to which instruction is used. POP and STORE instructions will go to control point OIT5, while LOAD and PUSH instructions will go to OIT11 instead. LOAD instructions will activate one more signal -- the WAIT signal, in OIT11.

POP and STORE will follow the same sequence of control points after OIT5 except on the error return from the MEMORY WRITE sequence in OIT6. OIT7 allows the POP instruction to correct what it has done to SCR when there is the necessity of an interrupt return.

Flip-Flops	Instructions			
	POP	STORE	PUSH	LOAD
POP	1	1	0	1
MSTOR	1	1	0	0
STOR	0	1	d	d

d = don't care

Figure 5.1.1.2/1

Operand Stack Instruction Sequences
POP, STORE, LOAD, and PUSH Control Step Flow Charts

5.1.1.3 DUP and SLUFF Control Logic

These sequences share the same control points OIT12 and OIT13, with the DUP sequence activating OIT12 and OIT13 conditionally. On return from the OS ENTRY, the two sequences will take up different paths.

The SPOP/S signal is used to set the POP flip-flop for OS ENTRY.

Operand Stack Instruction Sequences
DUP and SLUFF Control Step Flow Chart

5.1.1.4 XCH Control Logic

The SCH control sequence is used to exchange the top two cells in the operand stack. The first control point sets the POP flip-flop for the OS ENTRY control logic. It also sets STOR for the OS READ logic.

When the first OS READ sequence is begun, the X2 control signal must be turned on for the whole sequence. Note that since STOR = 1, the WAIT control signal to the SCR MOD sequence will automatically be turned on by the OS READ sequence called in control step OIT19. However since STOR is reset to 0 in OIT20 for the call to OS READ in control step OIT22, the WAIT signal must be turned on in this case to keep from wasting time modifying PR#0 if the SCR "wraps around".

Note that for the OS WRITE sequence call in control step OIT24, the LR and DR were loaded by the previous OS READ in step OIT22 and were not changed during the SCR MOD in step OIT23.

Operand Stack Instruction Sequences
XCH ENTRY Control Step Flow Chart

5.1.2 Multi-Cycle Instruction Sequences

The Multi-cycle Instructions consist of PUSHF, POPF, PUSHFR, POPFR, SCAN, SCANM, PACK, UNPACK, MOVE, TRANS and EDIT. These instructions are all 2 operand instructions except for the last 3 which have 3 operands and have the format shown in Figure 5.1.2/1.

Figure 5.1.2/1

In all of the two operand multi-cycle instructions the address phrase is a normal operand phrase which may be long or short. In the 2 operand instructions there are 2 short address phrases.

The count phrase, as shown in Figure 5.1.2/2 always is a short operand phrase whose tag field indicates the PR which is associated with the count. This PR may be "slashed" if desired. If the indirect bit of the count phrase is off, the count is contained in the PR itself; if it is ON, the count is either contained in the halfword indicated by the contents of the PR (replacement option) or the count is the sum or difference of the PR contents and the contents of the halfword (pre-addition or conditional subtraction option). Note that if conditional subtraction is used and fails, the multi-cycle instruction is not performed.

Figure 5.1.2/2

The continue bit is not used as such in the count phrase since in primitive instructions the number of phrases is known from the instruction type. This bit is not used as an immediate operand bit either, since counts are always immediate operands. Instead the continue bit of the count phrase is used to indicate whether the multi-cycle instruction will actually use the count. If it is off, the count in the PR is not used although it may have already been modified by the pre-scan operations. Instead the multi-cycle instruction is stopped as soon as a data cell with a flag on is processed. The count PR is not touched. If the count bit is on, the PR is decremented once for each byte processed and when the PR is zero the instruction stops.

The multi-cycle instructions are all used to manipulate large fields of data. Therefore in order to minimize the number of memory accesses, the data is taken from core in the form of double words whenever possible and then broken down into the proper cell size in the TP. Since the data may not begin on a double word boundary, however, a check for this is made at the beginning of the multi-cycle sequence and, if necessary, the data is brought into the TP in smaller chunks (usually bytes) until a double word is reached.

From this point on, the TP accesses double-words in memory to get most efficient use of each memory cycle. Finally at the very end, if the count has not ended on a double word boundary, or if a flag is found in the middle of a double word cell, the TP goes back to processing the data byte-by-byte until the end is reached.

As will be seen in the following sections the multi-cycle instruction sequences are divided up into 3 parts, an **initial part** which performs any operations peculiar to a particular multi-cycle instruction and which also sets up the main sequence, the main multi-cycle instruction sequence which takes care of getting the data, incrementing the count and pointer values, and storing the data, and the operation sequences which perform the operations on the data peculiar to the various multi-cycle instructions.

In the following sections the initial and final sequencing will be described and then the main multi-cycle sequencing. Next the special operations which must be performed during the main cycle for some of the instructions will be explained. Finally the special EDIT sequence is described.

EDIT does not use the standard multi-cycle sequence because it is too different from the other multi-cycle instructions.

5.1.2.1 Initial and Final Multi-Cycle Sequences

Each Multi-Cycle instruction has some sort of an initializing sequence which must be entered before the Main Multi-Cycle sequence is started. In addition the MOVE instruction has an exit sequence which must be performed after the main sequence is over.

In the cases of PUSHF, POPF, PUSHFR, POPFR, TRANS and UNPACK, the only thing which needs to be done is the setting of the control flip-flops for the main multi-cycle sequence. These settings are given on page 1 of the flowchart at the end of this section. The definitions of the various control flip-flops are as follows:

- MEM1: is on if the source data comes from core memory, off if it comes from the OS.
- MEM2: is on if the destination of the data is core memory, off if it goes to the OS.
- REV: is on if the data bytes must be reversed during the double-word access cycles.
- DECR: is on if the pointer register to core must be decreased instead of increased after each cycle.
- OP: is on if an operation of some sort must be performed during the main multi-cycle sequence.

The meaning of these control flip-flops will become more clear when the main multi-cycle sequence is explained.

For SCAN and SCANM, the initializing sequence consists of calling OS ENTRY so that the bit pattern (and possibly the mask) which are used in the comparison operation will be contained in the fast register portion of the Operand Stack. After this has been done the control flip-flops are set and the main sequence is begun.

The preliminary sequence for PACK is a bit more complicated, if the count option is used. If the end-on-flag option is

employed it is assumed that there are an even number of bytes in the input file which is to be packed. If the count option is used, however, the count may be odd, in which case the first byte to be packed will be loaded with 4 zeros followed by the packed byte. This condition is tested for by the preliminary sequence and the desired operations performed if necessary. Then the count is decremented by one and checked for zero. If it is not zero the control flip-flops are set and the main sequence is begun.

The MOVE instruction has the most complex preliminary sequence. The MOVE statement is executed in one of two ways depending on the state of the count and file pointer registers. If both file pointers and the counter PR are all multiples of 8 bytes, i.e. the file pointers point to double word boundaries and the count is a double word multiple, the MOVE is executed by using the double-word access sequence and simply transferring the data from the source file to the destination file.

If the pointer registers are not all multiples of 8, the Operand Stack is used as the destination file. This means that before the main sequence starts PRL3 is pushed to save its old value and then loaded with the value of the destination file pointer register. During the operation the data is moved from the input file to the OS. The Operand Stack hardware automatically buffers the data and stores it in the new file in double-word accesses. When the main sequence is over, the MOVE EXIT sequence is entered and the OS is then emptied and PRL3 is popped to return it to its original state before the instruction began.

The reason for the two different methods is that in the general case some form of buffering will be necessary to get the data from one file to the other since the initial boundaries will not necessarily coincide. The easiest way to do this is by means of the Operand Stack hardware. However this is slow to set up and one of the most common uses of the MOVE instruction will be when the supervisor relocates programs. Since in this case the counts and

boundries are "nice" the direct transfer method can be used to save time for the supervisor.

Multiple Cycle Instruction Sequences
Initial & Final Multi-Cycle Sequence Flow Charts

Multiple Cycle Instruction Sequences
Initial & Final Multi-Cycle Sequence Flow Charts

Multiple Cycle Instruction Sequences
Initial & Final Multi-Cycle Sequence Flow Charts

5.1.2.2 Main Multi-Cycle Sequence

As shown in the flow charts at the end of this section, the Main Multi-Cycle Sequence can be divided up into two parts, the part which deals with the initial and final ends of the field if they do not rest on double-word boundries, and the part which deals with the purely double -word accesses during the middle of the operation.

The first action of the multi-cycle sequence is to check to see if the pointer register being used for the operand address in core is pointing to a double-word boundry. If it is not, the non-double-word access sequence is entered and smaller cells are accessed until a double-word boundry is reached, at which time the double-word access sequence is entered. Usually the non-double-word sequence accesses bytes. However in the case of PACK instructions, half-words must be accessed so that a complete byte may be stored after the data has been packed. Also in the case of SCAN and SCANM instructions the data is accessed in sizes indicated by the CS variant bits. This is necessary in order to perform the compare operation. This also means that if the input field does not begin in a manner such that the cells eventually end on a double-word boundry, the control will never move into the double-word access sequence and thus never take advantage of the efficiency of double-word accessing.

If the core pointer register does not point to a double-word boundry, the next step is to see if the instruction being executed is one in which the PR must be decremented instead of incremented. If it is, this decrementation is done before the memory access in order to have the PR pointing to the front of the cell to be accessed.

Next the source data is obtained. This may come from either core or the Operand Stack, depending on the instruction being executed. PUSHF, PUSHFR, PACK, SCAN, SCANM, and MOVE all use the memory to contain the source field. POPF, POPFR, and UNPACK use the OS as the data source.

only handle double-words and one word of packed data will unpack into a double-word.

As soon as the data has been returned the flags are checked if the end-on-flag option is being used. If any flags are on the control transfers back to the non-double-word section, the PR must be re-incremented if it has previously been decremented on this cycle.

After the data flags have been checked, the data may need to be reversed. This is necessary if the PUSHFR or POPFR instructions are being performed. If so, a series of merging and shifting operations are used to reverse the data in the LR and DR.

Next the operations are performed, if necessary and also the data storage. Note that if the instruction is a PACK, the OS will only receive one word of data. There is also a special case for the MOVE instruction. If, as explained in the last section, the boundaries and counts of the MOVE instruction are all on double-word boundaries, the data transfers are made directly from one section of core to the other. In this case the NOSTK flip-flop will be set to "1" and this will cause the PR mentioned in the 3rd phrase to be used in the 2nd memory access. After the access, this PR must be incremented by 8.

After the memory access has been completed, the count is incremented, if the count option is being used, and the core pointer register is incremented if it has not previously been decremented. At this point the cycle begins again by checking the high order bits in the count pointer register.

After the data has been obtained the flags are checked if the end-on-flag option is being used. If the flags are not zero, the END flip-flop is set. If an operation is to be performed on the data, it is executed next. If a storage of the data is to occur it is performed after this.

After the data storage (or after the operation if no storage was performed) the count is decremented, if the count option is being used, and it is then compared to zero. It should be noted that the count is always stored in the Pointer Register whose name is stored in the tag portion of the last byte in the IR. If the count is zero, the multi-cycle operation is over and the post-execution operations are performed.

If the end-on-flag option was used, the END flip-flop is checked. If it is on, the operation is over.

If the operation is not over, the memory pointer is incremented if it has not been previously decremented. This ends the cycle. If this is the initial end of a field the control returns to the point where the core pointer register is checked to see if it is on a double-word boundary. If it is the final end of a field, the check is skipped and the cycle begins again with the decrement test.

The double-word access sequence is given on page 2 of the flow chart at the end of this section. It has the same general outline as the non-double-word sequence. However there are quite a few differences in the particulars.

In order to enter the double-word sequence, the high order portion of the count pointer register, i.e., all except the last three bits, must be non-zero. If this is true then there is at least one more double-word of data to be accessed. If the high order bits of the count are zero and the low order bits are non-zero then control returns to the non-double-word sequence and the final stage is begun. If both the high and low order bits are zero, the operation is over.

Assuming that there is another double-word of data to be accessed, the first thing done on entering the double-word sequence is to change the cell size indicator to double-word. If the PR must be decremented, this is done. Then the data access operation is performed. If the instruction is an UNPACK, only one word of data is obtained from the OS since the TP can

5.1.2.3 Multi-Cycle Operation Sequence

Four of the multi-cycle operations (PACK, UNPACK, SCAN and SCANM) must do an operation during the middle of the multi-cycle transfer operation. Therefore if any of these operations are to be performed, the OP control flip-flop is set and the proper operation is then performed after the data has been obtained from the source file.

The PACK operation consists basically of taking the last four bits of two consecutive bytes and packing them into one byte. This is done by means of special gates as shown in fig. 5.1.2.3/1. Using this setup up to 4 bytes may be packed at one time. In the SHORT OP ENTRY sequence only two data bytes are present so that this method is sufficient as is. In the LONG OP ENTRY there are 8 data bytes, 4 of which are in the LR. Therefore before the packing operation begins the contents of the LR are stored in the Spare Buffer Register (SBR). Then the last four bytes of the double-word (which are contained in the DR) are packing into the LR. After the LR is permuted left 2 bytes the remaining 4 data bytes can be packed to give one data word in packed format in the LR.

The UNPACK operation is the inverse of the PACK. The gates are arranged as shown in fig. 5.1.2.3/2. The data consists of one packed word in the LR. First the DR is cleared. Then the packed data in the first 2 bytes of the LR is gated to the last 4 bits in each byte of the DR. At the same time the first 4 bits of each byte of the DR are set to "0101". This unpacked data word is then stored in the SBR while the LR is permuted left 2 bytes and the remaining data unpacked. Finally the SBR is loaded into the LR and the LR and DR then contain one unpacked double-word.

The SCAN and SCANM instructions are simply multi-cycle forms of the TEST and TEST M instructions. Therefore all that must be done during the multi-cycle operation sequence is the

Figure 5.1.2.3/1 PACK Gating

Figure 5.1.2.3/2 UNPACK Gating

execution of these instructions for each cell of data. In the SHORT OP ENTRY this is very simple since only one cell of data will be present. In the LONG OP ENTRY, however, 1, 2, 4 or 8 cells may be present depending on whether the cell size is double-word, word, halfword, or byte. Since the DR will be needed in the comparison operation, the data stored in it is loaded into the SBR. Next the appropriate entry in the TEST sequence is chosen, depending on whether or not there is a mask in the Operand Stack. Since the TEST sequence leaves the data in the LR undisturbed when it is finished, it can be reused if CS is not a full-word.

When control returns from the TEST sequence, the count is decremented by 1. Since this sequence will not be entered if the count is zero and since only subtraction is employed on the count there will never be underflow. This means that the link part of the PR does not need to be gated into the new count value before it is restored to the pointer register and that means that the LR does not need to be disturbed during the count decrementation.

After the count has been decremented and checked for zero, the EQ indicator is checked. If it is on, the TEST sequence found a match and therefore the multi-cycle instruction is over. If there was no match, the PR is incremented next. Note that the method used here saves the old value of the LR and places it in the AR.

When the PR incrementation is done, the last 3 bits of the PR are checked. If they are all zero, the complete double-word of data has been checked. The control then returns to the multi-cycle sequence at the beginning of a new cycle since the count and PR have already been changed and since the SCAN and SCANM instructions do not store any data. If only the last 2 bits are zero, only one word of data has been checked. Therefore the word of data which has been stored in the SBR is loaded into the LR (right justified on the first cell of data ¹) and the TEST sequence is begun anew.

If neither of the above conditions hold there is still data being saved in the AR which has not been checked. Therefore the AR is gated into the LR with the proper permutation to put the next cell in the rightmost part of the LR. Then the TEST sequence is entered again.

5.1.2.4 Edit Sequences

Although the EDIT instruction is a multiple cycle operation, it operates considerably differently from the instructions previously discussed. It was decided therefore to use a completely separate sequence to perform this instruction since in order to add it to the main multiple cycle sequence too many changes would have to be made.

The EDIT instruction is a three operand instruction and therefore all three operands must be short. Its main purpose is to take a field of numeric data in packed format (source field) and, using a descriptive pattern which is found in the Operand Stack to modify its format, to load it into a second field in unpacked form (destination field). The format modification may consist of suppressing leading zeros and/or insignificant digits or adding certain other pertinent alphanumeric data, in order to prepare the source field data for some sort of output display.

A complete description of what EDIT is capable of doing is given in the programming manual and it will be assumed that the reader is familiar with it.

The overall operation of the EDIT sequence is as follows: The data field is read from memory one word at a time. After a data word is in the TP the pattern in the OS is popped one byte at a time. Then according to the state of the scan flip flop (SFF), the pattern byte and possibly the corresponding data byte, a blank, the unpacked form of the data byte or the pattern byte will be kept for storage in the destination field. After four pattern bytes (these are in unpacked form in the OS) have been examined, the resulting four bytes are stored in the destination field. When the word of data is exhausted a new word is read from core.

The instruction ends when the pattern count goes to zero, or, if there is no count, when a pattern byte is processed which has its flag bit turned on. Before the instruction finishes it stores all unstored bytes which have been processed into the destination field and increments the source and destination pointers so that they point to the first byte in their respective fields which was not processed. Note an even number of data digits is assumed.

The detailed flow chart is given at the end of this section. After the data has been accessed and a pattern byte obtained from the OS, the decoding is performed. The decoding is performed on the first byte in the LR and on the first or second four bits in the AR depending on the lowermost bit in the MCT counter. MCT is used to keep track of how many data digits have been used. The decoding is performed by the diode matrix shown in Figure A. (Not included at this time).

After decoding, the character to be stored is determined from the outputs of the decoder and the state of the SFF. Any changes in SFF are also made.

Note that if a data character is to be stored, it must first be unpacked and the four bit zone code, 0101, inserted in the leftmost positions in order to be in proper ASCII code. This is done using the same hardware which is used in the UNPACK instruction. After the unpacking, if the MCT is even the leftmost unpacked byte in the DR is placed in the LR. If the MCT is odd the second byte from the left is placed in the LR and the AR is permuted left one byte.

If a blank is to be stored it is set into the LR. If a data cell is then to be discarded, a shift of the AR is performed, provided the MCT is odd, and MCT is increased by 1.

After the character to be stored has been determined CT, which is used to keep track of how many characters have been processed, is incremented by 1. Then the SBR is placed in the DR which is gated into the LR with the 0th byte inhibited. This result is then permuted left one byte and restored in the SBR. After four characters have been stored the SBR will be full and it is therefore stored in the destination field. Before this is done however, the AR, which is holding source data, must be saved or else the data in it will be destroyed during the write sequence. Normally the SBR would be used for this purpose but since it contains the characters which are going to be written into core, the AR is stored in the OS. This will not involve any memory access since in order to have four characters to store, four pattern bytes must have been taken from the OS and therefore there is no danger of overflow.

After the characters are stored the top of the OS is read back into the AR.

At this point, if the count is zero or if there was no count and the flag on the last pattern byte was on, the instruction enters its ending sequence. Otherwise it checks the MCT and if it equals 8, reads in a new data word. Then it starts the cycle over.

The ending sequence first checks to see if $CT = 0$. If it does there is no unstored data to go in the destination field and only the source pointer needs to be updated to point to the byte following the last data byte to be used.

If $CT \neq 0$ the SBR must be shifted around until the 1st data byte to be stored is in the leftmost byte position. Then these bytes are stored in core one at a time. During this time the source and destination pointers are incremented to point to the proper position. The sequence is then finished.

5.1.3 Other Data Transfer Instruction Sequences

5.1.3.1 Assign Sequence Description

The assign sequence is quite simple. The basic assign sequence is simplified by the fact that most of the work is done by the immediate subsequences in the memory access logic. The assign sequence itself simply calls the read sequence, saves this result and then calls the write sequence. Note that one word of data can be kept in the LR since this will not be destroyed by the memory sequence. If there are two words of data the second word is kept in the SBR.

The operational and control logic flow charts are given at the end of this section.

Register and Core Data Transfer Sequences
ASSIGN Sequence Flow Chart

Register and Core Data Transfer Sequences
ASSIGN Control Step Flow Chart

5.1.4 List Processing Instruction Sequences

The list processing instructions in the TP include SL, SR, GET, PUT, INCL, INCR, DECL, and DECR. These instructions are intended to supply the basic operations for a variety of list processing systems. As such, they are quite general and can be used in a variety of ways with a variety of different types of lists. However, in devising the instructions, a certain "list processing philosophy" developed as to how to think about Illiac III list processing. The purpose of this section is to explain the ideas upon which this system is based so that future compiler or interpreter writers will understand how to take advantage of these instructions.

The basic list processing element is the "cell". A cell is an even number of contiguous bytes consisting of one or more pointer fields and an optional number of halfword data fields. The fields can be thought of as any size, but the total length must be an even number of bytes. This stipulation is consistent with the cell alignment convention that a halfword pointer will never overlap a halfword boundary, i.e. all the pointers must begin on an even byte.

The purpose of this cell format is to make the list processing system as flexible as possible so that a wide variety of LP system types can be utilized. As examples, bidirectional sequencing of lists such as those encountered in SLIP may be performed if the two sequence links point to the previous and succeeding cells. However by having both links point to succeeding cells, you can construct and scan binary trees. Stacks can be implemented by using only one pointer field to point to the next entry in the stack. Minor modifications to these structures will allow the construction of n-ary trees and circular lists.

The basic sources of storage in the Illiac III list processing instructions are the available space pointer registers. There is always at least one PR devoted to keeping track of available space (PR#14). In addition the programmer may dedicate other PR's to keeping track of available space if this is desired. The cells in available space may vary in size from 4 bytes on up. However a different PR must be used for each different cell size.

The cells for the structure are obtained using GET to obtain the needed cells and PUT to return them to Available Space when no longer needed. The structure can be built by using INCL and INCR, if bidirectional lists are being formed, by the slash conventions if stacks are being used, or by combinations of ASSIGN, SPECIFY, and other Illiac III commands for other types of structures.

In scanning through the various structures, a search is carried out using a pointer register as a "bug" (in the L^6 sense). The SR and SL instructions cause the PR to follow one of two possible pointers and loads the present location of the "bug" into the top of the OS.

In general, the OS is used to hold the parameters of the various list processing instructions.

It should be noted that the insert and delete instructions are by far the most specialized since they specifically assume that the list structure is to be a bi-directional list (e.g. SLIP lists). One reason for the direct implementation of these instructions was that without them SLIP implementation would be fairly inefficient when compared with Illiac III machine language. With these instructions, however, in combination with the other list processing instructions, it is virtually possible to implement a machine language level version of SLIP.

Figure 5.1.4 gives a first pass suggestion for a possible implementation of some common LISP 1.5 functions.

ILLIAC III LISP 1.5 Implementation

Basic LISP 1.5 functions	ILLIAC III representation
car[x]	ASSIGN(PR) .PRS, PRS==OS; SL PRS;
cdr[x]	ASSIGN(PR) .PRS, PRS==OS; SR PRS;
null[x]	TA;
eq[x,y]	CDRA;
cons[x,y]	POP(PL).PRS1; POP(PV).PRS1; GET PRAS; ASSIGN(PR) PRS2==OS, .PRS1;

PRS indicates scratch pointer register

PRAS indicates available space pointer register

All implementations assume function parameters are at top of OS. Parameters are destroyed and resulting address is pushed into O.S.

Figure 5.1.4

5.1.4.1 List Processing Sequence Descriptions

This section will describe the operation of the various list processing instructions in some detail. Before beginning, a small quirk in the instruction codes should be noted. In all of these instructions there are no cell size variants since the cell size is entirely determined by the available space pointer being used and does not concern the operation of these instructions. However whenever the OS is accessed in order to read or write cell pointers, it is done so on a half-word basis. Unfortunately the OS ENTRY sequence assumes that the cell size is determined by the Field Designator bits and there is no easy way to change this. In order to solve this problem the instruction codes for the list processing instructions were chosen so that the bits normally in the field designator positions always indicated a halfword cell. Thus no additional logic is needed.

The first four list processing instructions, SL, SR, GET, and PUT are fairly simple since they use the standard sequences developed for other instructions. The last four, however, INCL, INCR, DECL, and DECR are considerably more complicated.

SL and SR are used to "sequence" a pointer register along a list structure. They are identical except that when the PR is read out, if the instruction is a SR, the data must first be permuted so that the right link will appear in the LR left-justified.

The sequence begins by setting up the OS. Then the PR is read into the LR after which the LR left halfword is written into the OS. Finally this same halfword is gated to the DR right justified and used to access the cell being pointed to by the link. This data is then loaded into the PR.

Figure 5.1.4.1/1

Status of TP Registers During An Insert Instruction

GET is very simple since it uses the AS GET sequence (Section 4.3.1.2) to do most of the work. After setting up the OS, AS GET obtains a cell from the available space file and places the address of this in the DR right-justified. This address is then gated to the LR, left-justified and stored in the OS using OS WRITE.

PUT is also very simple since it uses the AS RESTORE sequence (Section 4.3.1.3). First the OS is set up and the half-word address is read out into the LR using OS READ. This is then gated to the DR and AS RESTORE is used to place the cell back on the free list in the available space file.

The INCL and INCR sequences are generally identical except for certain shifts which must be done for one version or the other. The sequence begins by preparing the OS with OS ENTRY and then loading the designated PR into the DR so that the link pointing to the cell on the side of the insertion is in the right half of the DR. The SBR link half is also loaded with this pointer. The SBR value half is loaded with the address of the pointer which points back to the cell originally "mirrored" by the PR. In the insert right case this is just the address of the next cell. For the insert left case however, it is the address of the next cell +2. Figure 5.1.4.1/1 contains a representation of the various register contents at various stages in the instruction.

Next a memory access is performed to obtain the actual pointer to the "mirrored" cell. This halfword is returned in the left half of the LR and after the access this data is stored in the SBR value field for an INCL or the SBR link field for an INCR instruction. Thus at this point the SBR contains the two links which must eventually be loaded into the inserted cell.

The next step is to change the forward and backward pointers on the left or right side depending on the instruction so that they point to the inserted cell instead of each other. The cell to the right or left is loaded by using its address which was previously stored in the SBR and read out before the value from the last memory access was read in. The cell pointed to by the PR is also accessed using its address stored in the SBR. The address which is written into the two links, i.e. the address of the inserted cell, is obtained by two separate accesses to the OS. Note that the PR is also modified to reflect the change in the cell it is "mirroring".

The final step is to load the two links in the SBR into the cell to be inserted. A third access must be made to the OS to obtain the address of the inserted cell. On this final access the cell is popped off.

The Delete Cell instructions, DECL and DECR, are not quite as complicated even though they can be thought of as the inverse of an Insert Cell instruction. The reason for this is that in an Insert Cell Instruction one figures out what data belongs in the two links of the inserted cell and then places it there; in the Delete Cell instructions, this is not necessary. Briefly, all the delete needs to do is change the two pointers to the cell being deleted. However, to do this it is necessary to read out the two links in the cell being deleted in order to find out the addresses of these pointers.

After setting up the OS using the OS ENTRY sequence, the address of the cell which is to be deleted is loaded into the LR left-justified and then pushed into the OS using OS WRITE. The states of the various registers during this instruction are given in Figure 5.1.4.1/2.

PR:

B	D
---	---

A:

	B
--	---

 B:

A	C
---	---

 C:

B	D
---	---

 D:

C	E
---	---

 E:

D	
---	--

LEFT

LR:

B	D
---	---

DR:

D	B
---	---

LR:

A	C
---	---

DR:

	A+2
--	-----

SBR:

C	A
---	---

RIGHT

D	B
---	---

 } AFTER OS WRITE

B	D
---	---

 } BEFORE 1st MEMORY ACCESS

C	E
---	---

C	E
---	---

C	E
---	---

 } BEFORE 2nd MEMORY ACCESS

LR:

A	C
---	---

DR:

A	C
---	---

PR:

A	D
---	---

E	C
---	---

	C+2
--	-----

B	E
---	---

 } BEFORE LAST MEMORY ACCESS

Figure 5.1.4.1/2

Status of TP Registers During a Delete Instruction

After the OS WRITE the address of the cell to be deleted is loaded into the DR, right-justified, and an access to that cell is made in order to determine the cells on either side of it. When the access is through this information, which is in the LR, is gated to the DR and SBR, in such a manner that the address of the cell after the deleted cell is right-justified in the DR. In the left delete case two must be added to this address in order to find the address of the pointer in that cell which originally pointed to the deleted cell.

Next, this same halfword pointer is written over with the address of the cell originally being mirrored by the PR. This in effect causes the deleted cell to be skipped in the list.

Finally, the remaining link to the deleted cell, i.e. the one in the cell currently being pointed to by the PR, must be corrected. Since the SBR contains both a pointer to this cell and one to the cell on the far side of the deleted cell, it is merely necessary to load the SBR into the LR and DR in such a way that the LR contains the pointer to the far side, left-justified and the DR contains the address of the halfword pointer in the cell mirrored by the PR, right-justified. For the right delete case this involves adding two to the cell address.

Just before the final memory access, the PR is loaded in the appropriate position with the address of the far cell so that upon completion of the memory access, the sequence is completed.

It should be noted that if one tries to delete a cell from a list with only one cell on it in the first place, there will be no change in the links of the cell on the list. In addition, the address of the cell will be placed in the OS. This may cause a

problem if the user continues to believe that his PR "bug" is pointing to a list since in fact the only cell previously belonging to that list will now be disposed of according to the instructions executed after the DECL or DECR.

The only way to detect for a double-linked list with only one cell in it is to check for both links being equal and also checking to see if the cell to which they point contains the same links. This involves a fair amount of added logic on the part of the delete cell instructions, and thus, for the present at least, such a test will not be performed. If it is found to be desirable, the test could be included and either a condition code set or an interrupt initiated.

9/2/69

Section 5.1.4.1 - 7/7

List Processing Sequences
SEQUENCE Flow Chart

List Processing Sequences
GETC Sequence Flow Chart

List Processing Sequences
PUTC Sequence Flow Chart

List Processing Sequences
INCL and INCR Sequence Flow Charts

List Processing Sequences
INCL and INCR Sequence Flow Charts

List Processing Sequences
DELETE Sequence Flow Chart

5.1.4.2 SL and SR Control Logic

The SL and SR control logic is used to sequence a pointer left or right. The signal LEFT, which is also used in the insert and delete cell instruction sequences is derived directly from the decoding of the instruction and determines the direction in which the sequence instruction will "move" the PR.

The control point logic used in these instructions is shared to a great extent by the other list processing control sequences. As a result the flow of control is fairly complicated. Figure 5.1.4.2 illustrates the implementation of the initial control steps in the list processing control sequences and shows how they are interconnected.

Note in particular that the sequence and delete control logic share control points LPT1 through LPT6, that the insert sequence shares LPT1 through LPT3, and that the GETC and PUTC sequences share LPT1 and LPT2. In addition the GETC sequence shares LPT4 with the sequence and delete control logic and the insert sequence shares LPT6.

It should also be noted that a variety of signals are turned on by control point LPT1 even though each list processing sequence only uses a subset of the total number of signals. Luckily with only two exceptions, those signals which are not used by a given sequence will not be affected by them. The exceptions are the signals which the POP control flip flop in OS ENTRY and the STOR control flip flop in OS READ. Since only the insert cell sequence needs STOR = 1, and since it does not execute OS READ until the middle of its sequence, the STOR flip flop is always set to 0 and later on in the insert cell sequence it is set back to 1 before the OS READ sequence is executed.

The POP control flip flop must be set using special select signals as shown in Figure 5.1.4.2.

9/15/71

Section 5.1.4.2 - 2/2

Figure 5.1.4.2 List Processing Sequence Control Logic
for Initial Part of Sequences

5.1.4.3 GETC and PUTC Control Logic

The GET and PUT Control Logic is designed to allow the programmer to have access to the hardware available space mechanism which was originally designed for the pointer stacks.

These two sequences both share LPT1 and LPT2 with the other list processing instructions and, in addition, GET shares LPT4.

List Processing Sequences
GETC Control Step Flow Chart

List Processing Sequences
PUTC Control Step Flow Chart

5.1.4.4 INCL and INCR Control Logic

INCL and INCR are used to insert cells in a double-linked list. This sequence is the longest of the list processing instructions in terms of both logic and time. The sequence makes use of the LEFT control signal, which is produced by the instruction decoder, to determine where to insert the cell.

The first three control points are shared with the sequence left and sequence right control logic. Control points LPT22 and LPT23, which increment the DR by two, appear twice in the insert sequence and also twice in the delete sequence. However note that they will only be used once during any one use of these sequences.

In control point LPT26, the STOR control flip flop for the OS READ sequence is set to 1 since it was incorrectly reset to 0 during control point LPT1.

The control points LPT36, LPT37 and LPT38 are shared with the delete instruction control system.

List Processing Sequences
INCL and INCR Control Step Flow Chart

List Processing Sequences
INCL and INCR Control Step Flow Chart

5.1.4.5 DECL and DECR Control Logic

The DECL and DECR control logic is used to delete a cell from a double-linked list. The sequence makes use of the LEFT signal which is generated by the instruction decoding logic to determine which cell next to the cell currently being looked at, will be deleted.

The sequence does not need much additional logic since it shares control points LPT1 through LPT6 with the sequence left or right control logic and control points LPT36 through LPT38 with the insert control logic. In fact, the only additional control points it uses are LPT39 and LPT40.

List Processing Sequences
DECL and DECR Control Step Flow Chart

List Processing Sequences
DECL and DECR Control Step Flow Chart

5.1.5 Unary Operation Instruction Sequences

The Unary Operations consist of those operations which only involve the top operand in the Operand Stack. These include: ONE, ZERO, NOT, COUNT, BIT, ABS, NEG, MNS, TA (test for algebraic zero), and CVL (convert to long fixed point). They can be divided up into logical operations (ONE, ZERO, NOT, COUNT and BIT), and algebraic operations (ABS, NEG, MNS, TA and CVL).

The last group is performed on algebraic numbers which may be in one of four number types:

- 1) Short Fixed - 2 bytes, 2's complement
- 2) Long Fixed - 4 bytes, 2's complement
- 3) Floating Point - 8 bytes, sign-magnitude
- 4) Decimal - 8 bytes, 16 4-bit ASCII characters

In each number type, the sign convention is different; to tell the sign of the number, a different bit must be checked for each number type.

In fixed point numbers in 2's complement representation, the sign bit is the leftmost bit in the number. If the numbers are right-justified in the TP registers, the sign bit for a short fixed point number will be in the 19th bit position from the left (counting 1 to 36) while the sign of a long fixed point number will be in the first position.

In a floating point number the sign of the magnitude is the leftmost bit of the double word. In all of the number types, if the sign bit is "1" the number is negative.

In a decimal number the ASCII sign character is located in the leftmost 4 bits of the double word. The code for "+" is 1010, while the code for "-" is 1011. Thus, to detect the sign of a decimal number, the fourth bit position of the most significant half of the double word is checked.

In the succeeding sections, the sequencing for the three groups of operations will be given along with detailed flow charts of the operations

5.1.5.1 Logical Unary Instructions

The Logical Unary instructions are those unary instructions which consist of purely logical operations. These include ONE and ZERO which replace the top cell in the OS with all ones or all zeroes, respectively; NOT which replaces the top cell of the OS with its one's complement and COUNT and BIT which determine respectively the number of ones in the top cell, or the position of the first one in the top cell.

5.1.5.1.1 Logical Unary Sequence Descriptions

The logical unary instructions consist of ONE, ZERO, NOT, COUNT and BIT. ONE stores all 1's in the top cell in the operand stack. The size of this cell is determined by the cell size code in the instruction. The sequence is implemented by sluffing off the top cell in the OS (see Section 5.1.1.1) and then forming all 1's on the DB. This can be accomplished by inhibiting the PLO/E line with the LRS/E signal. Since the DB normally inhibits the other permutation enable signals, by inhibiting PLO/E all the DB output lines are forced to "1". The DB is then gated to the LR and the OS WRITE sequence is initiated (Section 4.5.1.5). After this is finished, the sequence is done.

ZERO stores all 0's in the top cell of the OS. It works in the same manner as ONE except that the DB is loaded with all 0's by allowing PLO/E to remain on and inhibiting all of the permuter inputs.

NOT replaces the top cell of the OS with its 1's complement. This is done by reading out the complemented form from the OS and then immediately writing this result back into the stack.

COUNT counts the number of ones in the cell (excluding the cell) at the top of the Operand Stack. It is implemented basically by shifting the cell one bit at a time and counting the ones as they go by. The sequence begins by adjusting the Operand Stack, if necessary, so that the data is in the fast register portion and also so that there will be room for the halfword count. Note that if the cell size is only a byte, the stack must be checked for both overflow and underflow.

Next, one word is popped from the top of the stack. If the cell is a double word the second half is saved in the SBR. The counters are all reset. The BYCT is used to count the number of bytes which have been processed while the BICT is used to count the number of bits in the byte which have been processed.

The checking cycle begins by incrementing BICT. Then if the leftmost bit position of the LR is 1, the ONCT is incremented. If BICT is zero the last bit in the byte has been tested. Then the BYCT is

incremented.

Since we are at the end of a byte we must see if we have finished inspecting the cell. The BYCT is checked for various counts depending on the cell size. If the proper number of bytes have been inspected, the count in the ONCT is loaded into the LR so as to form a halfword, left justified. This is written into the top of the stack and the instruction is then finished.

If the cell size is a double word and BYCT = 4, then the second half of the cell is read out of the SBR into the LR, the BYCT is incremented and the cycle begins again.

If the end of the byte does not signify the end of the cell, or if a new word is not accessed the BYCT is incremented and the data is shifted one bit position to the left before beginning the cycle again.

The final step in the cycle, if the data bit was not one and the end of the byte is not reached, is to shift the data bits (not the flags) one bit position to the left. Then the cycle begins again.

BIT is actually quite similar to COUNT and in fact uses the same sequence with only minor changes. It delivers the address of the first bit in the top cell of the OS and pushes it into the stack in a halfword cell. In BIT the BYCT and BICT are still used to count the number of bytes and bits, respectively, which have been processed. However the ONCT is used to keep track of the position being checked. Thus it is incremented each time a bit is checked and when a data bit which is "1" is discovered, the contents of the ONCT are pushed into the top of the Operand Stack.

The flow charts for these five instructions are given at the end of this section.

Logical Unary Instructions
ONE Sequence Flow Chart

Logical Unary Instructions
ZERO Sequence Flow Chart

Logical Unary Instructions
NOT Sequence Flow Chart

5.1.5.1.2 ONE, ZERO and NOT Control Logic

These sequences are very straightforward. ONE and ZERO share the same sequence except in OIT39 where the ONE instruction generates all ones on the DB bus while the ZERO generates only zeroes.

NOT shares the same control points, OIT35, OIT36 and OIT40, with ONE and ZERO. It also turns on OIT35x conditionally. In OIT37, it reads out the negative form of the top cell in OS. Then it writes this negative form into OS again in OIT40.

Logical Unary Instructions
ONE and ZERO Control Step Flow Charts

Logical Unary Instructions
NOT Control Step Flow Chart

5.1.5.1.3 COUNT and BIT Control Logic

The COUNT and BIT control logic, although not very long in terms of control points is fairly complicated because of the loop in which the data bits are inspected. In order to make this loop as fast as possible, a great deal of it is done in parallel and therefore the logic attached to the task signals is fairly complex.

The first part of the sequence, which sets the counters and flip-flops, is straightforward. Note that the BYCT, BICT and possibly the ONCT are incremented during the execution of the OS ENTRY sequence.

The main loop in the COUNT-BIT sequence consists of 3 control steps: OIT49, OIT50 and OIT51. The first of these is especially complicated since it incorporates most of the logic in the loop in the flow chart shown in Section 5.1.5.1.1.

The basic operation of the loop is as follows: before the first step the LR has been properly loaded with valid data. Thus, during this step the LR is gated to the DR in preparation for the next left shift and simultaneously is checked for a one in its left-most position, LR1. Also if the BICT is zero, the current bit being checked is the last one in a byte. In this case 2 operations are performed simultaneously: 1) The BYCT counter is checked to see if this is the last byte of the cell, and 2) the BYCT is also checked to see if, in the case of a double word cell, it is necessary to reload the LR with the second half of the cell.

The second step in the sequence consists of performing a left shift by one bit which gates the data in the DR back into the LR, and also performing any counter incrementations which were indicated by the results of step one. These indications are made by the use of flip-flops.

The third step is to reset the counter indicator flip-flops.

The control point execution in the loop is generally controlled by what happens during the checking portion of first step in the loop. In order to "remember" what happened after the task signal has been turned off, two flip-flops are used: ENDS, which is turned on if the end of the sequence has been reached and, GSBR, which is turned on if the SBR must be loaded into the LR.

Figure 5.1.5.1.3 - Task Logic For
1st Task in Count-Bit Loop

The counter incrementation of BYCT is also controlled by a flip-flop, BYIN. There is no need for a flip-flop for BICT since it is always incremented.

The logic which is used in conjunction with the task signal of the first step in the loop is shown in Figure 5.1.5.1.3. Note that in the most common case, when the bit being examined is not the last bit in the byte, all that is done is to gate the LR to the DR, increment ONCT and set ENDS according to the state of LRI and whether the instruction is a BIT or a COUNT.

When the sequence finally comes to an end the count must be loaded from the ONCT into the LR so that it may be written into the OS as a half-word count. This is done by turning on the ONP/E signal, which will gate the outputs of ONCT into the constant generation lines of the permuter.

5.1.5.2 Algebraic Unary Instructions

The Numeric Unary Instructions consist of the sign change instructions, ABS, NEG, and MNS, the test algebraic instruction TA which was mentioned previously in Section 4.3.2 on the algebraic logical compare instructions, and the conversion to long fixed point, CVL. Their operation is described in the following sections.

5.1.5.2.1 Algebraic Unary Sequence Descriptions

The Algebraic Sign Change Operations are those which involve a possible change in sign for the top operand in the Operand Stack (OS), i.e. ABS, NEG, and MNS. In ABS, the top cell of the OS is replaced by its absolute value, while in MNS it is replaced by the negative of the absolute value. NEG replaces the top cell with the negative of its value. The object of the sign change sequence is, therefore, first to see if it will be necessary to change the sign at all, and second, if it is necessary, to make the change and replace the cell in the OS.

The sequence begins by setting up the OS for a read operation. If the cell is in fixed point format, the 1's complement output from the OPERAND STACK is used. This simplifies changing the sign later on if this becomes necessary. The other number types are read out of the OS in "positive" form. However, since the decimal and floating point numbers both use a sign-magnitude format and since in both cases the sign is in the left-most word of the double-word cell in the OS, only this one word need be pulled out of the stack to determine the sign and change it, if necessary. When the OS READ sequence is finished, the desired word is contained on the DB, left-justified.

The next step depends on the number type. If the cell is fixed point, its sign is checked by looking at the first bit position on the DB. Since OS READ returns all cells left-justified, this will always be the sign position whether the cell is long or short fixed point. Remembering that we have taken the complement of the OS contents, we know that if the sign on the DB is "+" and a MNS instruction is to be performed no further action need be done, and the sequence is finished. The same is true if the DB sign is "-" and instruction is ABS. However, if the NEG instruction is to be performed, or if the DB sign is "-" and the instruction is MNS, or the DB sign is "+" and the instruction is ABS then the sign of the operand must be changed.

This is essentially done by taking the 1's complement output from the OS and using the 32 bit adder to inject a low-order carry and convert the number to the 2's complement negative. In doing this, of course, the DB must be set to zero so its contents will not affect the addition.

The procedure differs slightly for the long and short fixed point cases. For the long fixed point, the DB is gated to the DR, the DB is then set to zero by disconnecting all of its inputs, and the carry is injected into the adder. When the carry propagation is finished the AR (the adder output) will contain the negative 2's complement of the original cell at the top of the OS. The AR is then gated into the LR to be written back into the OS by OS WRITE.

For the short fixed point case, the OS half-word complement must be right-justified before it can be gated to the DR. Since the BROS is still attached to the Permuter all that needs to be done is to set the Permutation control for right justification. This is done by setting DBRJ = 1. At this point IBO and IBI are also set to 1 to inhibit the leftmost 2 bytes. The DB is then gated to the DR, after which it can be set to zero and the low-order carry injected. Finally the AR is gated back to the LR, shifting left 2 bytes as it goes through the Permuter.

The final part of the fixed point sequence is writing the LR back into the OS.

As stated earlier in the case of floating point and decimal numbers only the most significant word (i.e. the leftmost) is read out of the OS. The procedure for changing the sign in these cases is very simple since it only involves changing 1 bit, although the bit is in a different location in the two number types.

For non-fixed point numbers, after the OS READ Sequence has "returned" the DB is gated to the AR where the sign is checked. If the instruction is a MNS or ABS the sign is automatically forced to be "-" or "+" respectively and the sequence is finished. Otherwise the sign must be checked and then "inverted". If the cell is floating point the sign bit is in the first position, while if the cell is in BCD the sign code is in the first 4 bits. However since the decimal code

for "+" is 1010 and the code for "-" is 1011, only the fourth bit position need be changed when making a sign change.

The Test Algebraic Instruction, TA, is essentially a comparison instruction, and thus the same basic hardware can be used as in the two operand compare sequence in Section 4.3.3.2. In addition, since there is no arithmetic manipulation necessary, the floating point and Hollerith number types can also be tested with only a small additional amount of logic. This logic is used to detect the value of the sign in these number types, since their sign representation is not the same as in fixed point.

The basic idea in the sequence is to load the desired data into the AR and then check for GT, LT, and EQ just as in the compare operation. If the AR contains all zeroes the EQ flip-flop is set, which causes the GT and LT flip-flops to be reset to zero. The GT and LT flip-flops are originally set according to the sign of the number. In floating point and decimal representations, the numbers take up two words of storage so that it is necessary to load both words into the AR in succession in order to be sure if EQ is on. The sequences for each number type are given below.

A fixed point number is tested in one cycle: the cell is gated right-justified into the AR. The EQ, LT, and GT indicators are set, and then the LT and GT conditionally reset as outlined above.

Floating point and decimal numbers are tested in two cycles. For floating point, the left half (of the double word cell) is gated into the AR. The EQ, LT, and GT flip-flops are then set. If EQ = '0' (not equal to zero), the test is complete. If EQ = '1' (equal to zero), the righthalf cell is gated into the AR and tested for all bits equal to zero. Finally, the LT and GT are conditionally reset. A floating point number is zero only if all bits (sign, exponent, mantissa) are zero. If the sign is "-", LT will be set even though all other bits may be "0".

Decimal numbers are similarly treated. Here, however, the right half cell is first gated into the AR (Since it contains the sign code). Again, the EQ, LT, and GT indicators are set. The low-order 4 bits are prevented from participating in the EQ test since they contain a nonzero sign code. The GT and LT are set from bit 35. If EQ = '1' the left half cell is then gated to the AR; if not, the test is finished. If EQ is '1' all of the left cell bits are checked for zero. LT and GT are then conditionally reset.

The CVL sequence is performed by the TP if the number type is fixed point. If it is floating point or decimal, the data is sent to the AU.

The basic CVL sequence is very simple. Since a long fixed point number needs no conversion, this case results in a NOP. For short fixed point numbers, it is merely a case of reading out the half-word of the data, extending its sign bit, and rewriting the full word of data back into the OS.

Algebraic Unary Instructions
TA Sequence Flow Chart

Algebraic Unary Instructions

CVL Sequence Flow Chart

5.1.5.2.2 ABS, NEG and MNS Control Logic

The ABS, NEG, and MNS instructions all share the same basic control sequence with slight modifications for the various instruction and data types.

The first three control steps load the necessary data onto the distribution bus, DB. Note that in the case of non-fixed point data the leftmost half of the double word is obtained. In order to do this it is necessary to reset the cell size generator gate selector so that the cell size can be controlled by the CCSWE signal. However, when the OS READ sequence is completed, the number type indicators will all be off since their gate has not been selected. For this reason a special flip-flop must be set up to control the flow of control after AUT3. This logic is shown in Figure 5.1.5.2.2/1.

If the number type is not fixed point the change in sign is performed by complementing one bit in the data. The logic which performs the actual operation is given in Drawing 05-3 of the Illiac III Main Frame Drawing book. In order to force a sign to "+", a "0" signal is dot-or'ed with either AR1 or AR4 which will cause the sign of the contents of the AR to be plus when it is read back into the LR. In order to force a sign to "-", a "0" signal is dot-or'ed with either AR1 or AR4. This causes the AR1 or AR4 outputs to 1 (see 1018-237 card schematic) and the sign of the AR contents will thus be "-" when the AR is gated to the LR. The control signals for this logic are listed in Figure 5.1.5.2.2/2 and produced by control steps AUT5 and AUT6 as shown in Figure 5.1.5.2.2/3.

If the number type is fixed point the advance out signal from AUT3 is used to drive some logic which determines if a change in sign is necessary and if so picks one of the two "versions" of the sequence AUT7, AUT8, and AUT9. These two versions are almost identical except that one is used for short fixed point numbers and includes some extra shifting in AUT7 and AUT9 so that the conversion from 1's complement to 2's complement will be correct.

Note that AU9 is also shared with the non-fixed point sequence.

Figure 5.1.5.2.2/1 - Storage Logic for Number Type in Algebraic Unary Instruction Control Logic

9/10/69

Section 5.1.5.2.2 - 2/4

UNARY Control Signals:

ABSF/E = Absolute value (floating point)

ABSH/E = Absolute value (BCD - Hollerith code - decimal number)

MNSF/E = Set minus (floating point)

MNSH/E = Set minus (BCD - Hollerith code - decimal numbers)

NGF1/E = Change sign (negate) (floating point), cycle 1

NGF2/E = Change sign (negate) (floating point), cycle 2

NGH1/E = Change sign (negate) (BCD - Hollerith code-decimal numbers), cycle 1

NGH2/E = Change sign (negate) (BCD - Hollerith code-decimal numbers), cycle 2

UNARY Internal Signals:

Simulation Name	Logic Book Name	Description
AR	ARj	Arithmetic Register
NEG	None	= 1 if number to be operated on is minus

Figure 5.1.5.2.2/2

Figure 5.1.5.2.2/3 - Sign Change Gate Generation Logic for Non-Fixed Point Numbers

Algebraic Unary Instructions
ABS, NEG, and MNS Control Step Flow Chart

5.1.5.2.3 TA Control Logic

The Test Algebraic control logic implementation is very straightforward. It should be noted that the $NWTOS = 1$ option is not used in OS READ because $STOR = 1$ and both of these may not be 1 during the same OS READ sequence.

Algebraic Unary Instructions
TEST ALGEBRAIC Control Step Flow Chart

5.1.5.2.4 CVL Control Logic

The Convert to Long Fixed Point instruction, CVL, is very straightforward. If the number type is floating point or decimal, an exit into the AU sequence is performed.

Algebraic Unary Instructions
CVL Control Step Flow Chart

5.1.6 Boolean/Shift Instruction Sequences

The Boolean/Shift instructions are divided into three groups: the Boolean Operation instructions, AND, OR, XOR, and EQV; the Boolean Transfer instructions, SET and RESET and the Shift instructions, LS and RS. The operation of these instructions is explained in the following sections.

5.1.6.1 Boolean Operation Instructions

5.1.6.1.1 Boolean Operation Sequence Descriptions

The basic Boolean Operation sequence is quite straightforward since most of the work is done in the boolean hardware (See Section 2.8.1.1). This sequence executes the AND, OR, XOR and EQ instructions. The sequence fetches operands from the Operand Stack and place one in the DR and the other on the DB. Then by activating the proper control lines the correct boolean result is produced in the LR by the Boolean Logic circuits, and this result can then be stored back in the OS. A test for zero is also made on the result and the EQ flip-flop set accordingly.

There are two complications in this scheme: one for the case of the XOR instruction and the other for double word cells. As explained in Section 2.8.1.1, the XOR function is formed by using the same hardware as in the EQV function but complementing one of the inputs, i.e.

$$X \oplus \bar{Y} = X \oplus Y$$

Thus, if the XOR instruction is to be performed, the first operand is brought out of the Operand Stack in 1's complement form by setting the NEG control line to the OS READ sequence to "1".

In the case of a double word cell the Operand Stack reading becomes more difficult, since the Boolean Logic can only work on one-word cells or less. Therefore, after the first cell has been read out of the OS, CSD is checked to see if the cell is a double word. If it is, the LR and DR contain the first and second halves respectively of that word.

In this case the LR is stored in the Spare Buffer Register (SBR) and the second half of the second operand is read out of the OS into the LR by the OS READ sequence. At this point the DB also contains the second half of the cell since the Permuter input is still connected. Thus since the DR contains the second half of the first operand, the boolean operation can be performed on these two words and the result will appear in the LR. This result is stored in the AR to make room for the second half of the boolean sequence and to allow the first part of the result to be compared to zero.

The next step is to read out the rest of the 2nd data cell from the operand stack. Note that a word cell size is being used and that the OS pointer has already been decremented by 4 bytes during the last OS READ. Therefore by again using the STORE = 0 option in the OS READ sequence we will get the left-hand side of the second data cell in the OS and the pointer will be decremented by another four bytes. Note also that by setting NWTOS to 1, the data read from the OS will only be placed on the DB and not gated into the LR.

Once the data is in the LR, the second half of the boolean operation can be performed. Its result will be returned to the LR. At this point, if the cell is a double word, the LR is gated to the DR and the DB is loaded with all zeros so that the flags can be compared to zero. When this is completed the DR is loaded with the results from the first half of the operation which were stored in the AR.

At this point if the EQ flip-flop was set to 0 by the comparison of the first partial result there is no need to look at the second partial result. If EQ = 1, however, the second partial result is loaded into the AR to be compared to zero. The same procedure is then used with the FM flip-flop.

If the data cells are not double words the same basic steps are performed except that they are only performed once.

The final operation is to write the result back into the OS, if this operation has not been inhibited by setting SKWT = 1.

The order of removal of data from the OS is shown graphically in Figure 5.1.6.1. The flowcharts from the boolean operations are given at the end of this section.

Beginning of operation

End of first OS access

End of second OS access
(if double word)

End of third OS access
(if double word)

After OS WRITE

Figure 5.1.6.1

Boolean Operation Instructions
Boolean Operation Sequence Flow Chart

Boolean Operation Instructions
Boolean Operation Sequence Flow Chart

5.1.6.1.2 AND, OR, XOR and EQV Control Logic

This control logic executes the four boolean operations, AND, OR, XOR and EQV. There is one control signal, SKWT, which, when 1, indicates that the final result should remain in the LR (and possibly the DR) and should not be written into the OS. This also causes a return to be made to a calling control point.

Except for the loop, the logic is not very tricky. All four instructions share the same initial control step except that XOR must set NEG to 1 instead of 0. Entry A is used as an entry point for SET and RESET (see Section 5.1.6.2).

After the first boolean operation, the process must be repeated again if the cell size is double word. This involves repeating the OS READ sequence, the boolean operation, and the loading of the result into the AR and testing it. Note however that before loading the AR with the leftmost half of the result, the rightmost half, currently in the AR must be stored in the DR.

Implementation of the logic for the loop is shown in Figure 5.1.6.1.2. Since the control points are all set before the sequence begins, on the first pass, if the cell size is a double word, control point BST10 will be executed after BST9, since before its execution its $\bar{A}_0$ signal will be 1. On the second pass the $\bar{A}_0$ signal of BST10 will be 0 and thus either BST12 or the logic after it will be used, depending on the state of the EQ and FM flip flops.

In setting up the control points for the second pass in the loop, note that BST9 sets control points BST4, BST5, BST6 and BST7 while BST6 sets control points BST8 and BST9. Control point BST10 is only used once in the loop and therefore does not need to be set. This allows the A_0 and $\bar{A}_0$ signals from BST10 to be used to control the flow of operations during the loop.

Section 5.1.6.1.2 - 2/2

Boolean Operation Instructions
AND, OR, XOR, EQV Control Step Flow Chart

Boolean Operation Instructions
AND, OR, XOR, EQV Control Step Flow Chart

5.1.6.2 Boolean Transfer Instructions

5.1.6.2.1 Boolean Transfer Sequence Descriptions

The Boolean Transfer sequences are used to execute the SET and RESET instructions. As shown in the flowchart at the end of this section, they are quite straightforward.

The SET instruction uses a mask to indicate which bits of the cell pointed to by the operand address are to be set to "1". To perform this operation, the mask in the $\emptyset$ S and the data at the operand address are OR'ed together and then stored back in the original address.

The RESET instruction uses a mask whose 1's determine the bit positions in the data at the operand address which are to be reset to "0". To perform this operation, the complement of the mask is AND'ed with the data and then restored in the original operand address.

SKWT := 1 causes the OS WRITE sequence to be left out after the boolean operation is finished

Boolean Transfer Instruction

Boolean Transfer Sequence Flow Chart

Section 5.1.6.2.1

5.1.6.2.2 SET and RESET Control Logic

The SET and RESET sequences are used to change cells in memory based on a mask contained in the Operand Stack. These two instructions share the same sequence which in turn utilizes the boolean operation sequence.

Boolean Transfer Instruction
SET and RESET Control Step Flow Chart

5.1.6.3 Shift Instructions

5.1.6.3.1 Shift Sequence Description

The Shift sequence is used to shift an operand which is at the top of the Operand Stack, left or right on a bit basis, using a count specified by a pointer register. The count is loaded into the M counter (MCT) which is used to count the number of bits shifted.

The sequence itself works in three stages. The first is the setup stage. The MCT is loaded by gating the PR contents into the LR and gating the LR to the MCT. If the count is not zero, the OS is then prepared and the top cell is read out. At this point, if the count is ≥ 32 the cell is set to zero and written back into the stack. Otherwise the second stage begins.

It should be remembered that the shift instructions cannot be executed on double word cells. The reason for this is that there is no double-word shift register in the TP and it would take too much hardware to adapt the 32-bit shift logic for double words.

The second stage consists of a shift of the data on a byte basis. It will only occur if the count is ≥ 8 . In this case the permuter is used to shift the data the largest possible number of bytes without exceeding the bit count. This byte shift is performed in one permutation regardless of the number of bytes shifted.

The third stage consists of a maximum of 7 bit shifts which are performed one at a time while the MCT is decremented. As soon as the lower order three bits of the counter are zero the shift is completed and the result in the LR is written back into the OS.

Shift Instructions
SHIFT Sequence Flow Chart

5.1.6.3.2 LS and RS Control Logic

The LS and RS control logic is fairly simple. The control of the sequence is maintained in large part by three control signals, CTLHZ, CTMHZ, and CTMLZ. The CTMHZ (count of the M counter high 2 bits zero) and the CTMLZ (count of the M counter low 3 bits zero) signals are generated by the logic associated with the M counter in Drawing 17-1, and indicate that the high order 2 bits or the low-order 3 bits of the M counter, respectively, are all zero. The CTLHZ (count of the LR high order bits zero) signal is generated by the IC logic shown in Figure 5.1.6.3.2/1 and indicates that LR bits 19 through 30 are zero. These are the high order 11 bits of the 16 bit count.

The loop used in the shift is a very tight loop involving two control points BST24 and BST25. Figure 5.1.6.3.2/2 shows a possible logic implementation. Note that the flip-flops are used in parallel with the control points to keep the control point on, even while the immediately previous one is being reset. This eliminates the problem of having the task signal turn off because the advance in signal deactivates too soon. The extra flip-flops can be reset by the task signal of the immediately preceding control point. Note also, however, that the flip-flop coming immediately after BST24 must also be reset by the master set signal for the sequence. Otherwise, BST25 would "fire" as soon as it was set and released since the flip-flop would still be on.

Figure 5.1.6.3.2/1 - Generation of CTLHZ

9/10/69

Section 5.1.6.3.2 - 2/3

Figure 5.1.6.3.2/2
Shift Instruction Loop Control

Shift Instructions
LS and RS Control Step Flow Chart

5.1.7 Algebraic/Logical Compare Instruction Sequences

The logic controlled by this section is used to accomplish algebraic and logical compares (greater than, equal to, and less than), the equal to zero test, and flag match compares. The instructions realized are CPRA, CPRL, TEST, SCAN and TA. Since SCAN is a multi-cycle instruction it will be explained in Section 5.1.2. TA is explained in the section on algebraic unary operations, Section 5.1.5.2.1.

In addition to these instructions certain other sequences make use of the compare logic. However in these cases it is a case of using the logic described in Section 2.9 and the actual control logic described in the following sections is not used.

The basic compare operation is performed by subtracting one of the given operands from the other and then comparing signs to set GT and LT and checking for a zero result to set EQ. The TP logic can only handle long and short fixed point numbers. BCD and floating point numbers are routed to the AU.

It should be noted that since the comparison is based on a subtraction, and the TP contains only a 32 bit adder-subtractor it is not possible to do a double word logical comparison. The hardware necessary for performing two subtractions and remembering the carry between them appears to be too costly. Floating point and decimal comparisons are done by the AU.

5.1.7.1 Basic Compare Sequence

5.1.7.1.1 A/L Compare Sequence Description

The basic TP Compare Sequence is used by the various compare instructions. It incorporates the common operations for these sequences and begins with the assumption that the DR has been loaded with one of the operands in non-complemented form. It also assumes that the other operand is in the Operand Stack. The result is a comparison between the contents of the DR and the contents of the top of the OS.

After resetting GT, LT, and EQ to 0, the first operation is to read the second operand out of the OS in 1's complement form. There is a provision to allow this to be the first or the second operand; the actual decision being made by the calling sequence by turning on the X2 control signal before entry if the 2nd cell is desired.

The control then makes a check to see if the signs of the operands are different (i.e. if the signs of the DR and DB are the same) because if this is true, the comparison can be easily made without resorting to subtraction.

If the signs of the operands are the same, the DR and DB are added (effectively subtracting the operands) and the result is placed in the AR. At this point the logic described in Section 2.9 is used to make either an algebraic or logical comparison. The algebraic comparison treats the data as signed 2's complement numbers while the logical comparison treats them as positive unsigned numbers.

After the first cycle of the A/L compare logic, the EQ will have been set if the difference stored in the AR was zero. However the GT and LT will also have been set on the basis of the signs alone. Therefore, if the EQ is on, a second cycle is necessary, in order to reset the GT and LT flip-flops to zero.

Note that the A/L compare logic has also set the flag match flip-flop, FM, according to whether the flags on the two operands are the same or not.

The OS pointer is unchanged by this sequence.

5.1.7.1.2 A/L Compare Control Logic

The A/L COMPARE sequence is used to do the actual comparison between two operands. The first must be loaded into the DR before the sequence begins. The second is assumed to be in the Operand Stack. The only control flip-flop for the sequence is ALG which must be set to 1 if the sequence is to perform an algebraic comparison and 0 if it is to perform a logical one.

In the A/L Compare Control Logic the logic is fairly straightforward. Control Point CMT1 is used to reset the flip-flop which holds the data on the DB after the previous OS READ. This is needed in the case of some calls to this sequence but will not harm anything when it is not needed.

Note that X2 may have been turned on by the calling sequence for the duration of the OS READ, in which case it must be reset, this being done in control point CMT3 after the OS READ has finished. Control Point CMT3 also enables the sign comparison which checks for oppositely signed operands (the signs of the DB and DR must be the same since the DB is the compliment of the operand). The actual logic used is shown in Figure 5.1.7.1.2/1.

Basic Control Sequence
A/L Compare Control Step Flow Chart

5.1.7.2 A/L Compare Instructions

The instructions which will be covered in this section are Compare Logical (CPRL), Compare Algebraic (CPRA), Test (TEST) and Test Masked, (TESTM). As mentioned earlier, Test Algebraic (TA), is discussed in Section 5.1.5.2 and the scan instruction (SCAN), is discussed in Section 5.1.2.X.

9/2/69

Section 5.1.7.2 - 1/1

5.1.7.2.1 A/L Compare Instruction Sequence Descriptions

CPRA and CPRL are compare instructions which compare the top two operands in the operand stack (OS). The next-to-top operand is compared to the top operand, i.e. the next-to-top operand ($\frac{>}{<}$) the top operand.

These two instructions are executed by the TP if the number type is fixed point and by the AU if it is floating point or BCD. In the latter case control is immediately passed to the AU ENTRY sequence (Section 5.2.1.2) which accesses the data and sends it to the AU. A return word from the AU carries in its flag bits information regarding the compare results, as well as other data on the validity of the transmitted cells and other operation results.

Words returned from the AU are gated from the cable terminators (CT) first into the LR, then into the DR if a second word is received. In the present case two words are received. The position and meaning of the flag bits are as follows:

LR9	GT	(greater than)
LR18	EQ	(equal)
LR27	LT	(less than)
LR36	OV	(overflow)
DR9	FM	(flag match)
DR18	UN	(underflow)
DR27	LS	(loss of significance)

The flags are then gated to their respective indicators.

If the number type is fixed point, both CPRA and CPRL use the same sequence except for the fact that CPRA sets the ALG flip-flop to 1 and CPRL sets it to 0. Both instructions begin by using the OS ENTRY sequence to prepare the OS hardware registers. Next the top operand in the stack is gated right justified into the DR in "true" form using the OS READ sequence in the NWTOS = 1 mode. Note that this OS READ pops off the top cell in the stack. Finally the A/L COMPARE sequence is executed after which the instruction is completed.

TEST and TESTM are instructions which compare data in memory with data in the OS. TEST is an ordinary comparison while TESTM utilizes the next-to-top cell as a mask which is AND'ed with the data before it is compared to the top cell. Neither instruction changes the OS.

Both of these instructions begin by preparing the OS hardware with OS ENTRY and then making a memory access to get the desired data. The TESTM sequence uses an OS READ at this point to read out the next-to-top cell and then AND's it with the data. Both sequences then store the resulting data in the DR right justified, reset ALG to 0, and execute the A/L COMPARE sequence after which they are finished.

The flowcharts for these instructions are shown at the end of this section.

A/L Compare Instructions

CPRA and CPRL Sequence Flow Chart

A/L Sompore Instructions
TEST and TESTM Sequence Flow Chart

5.1.7.2.2 CPRA and CPRL Control Logic

This sequence is very straightforward. At control point CMT8 the ALG flip-flop is set to 1 for CPRA and 0 for CPRL. Note also that the flip-flop which holds the data on the DB after the OS READ execution in CMT10 is reset in the first control step of A/L COMPARE in CMT12.

A/L Compare Instruction
CPRA and CPRL Control Step Flow Chart

5.1.7.2.3 TEST and TESTM Control Logic

The Control Logic for TEST and TESTM is very straightforward. Note that X2 must be turned on for the duration of the OS READ sequence called by CMT17. Also note that the last control step is shared with the CPRA and CPRL sequence.

A/L Compare Instruction
TEST and TESTM Control Step Flow Chart

5.1.8 TP Utility Instruction Sequences

5.1.8.1 Specify Instruction

5.1.8.1.1 Specify Sequence Description

The SPECIFY instruction is not like the rest of the TP primitive instructions in that it does not use the normal phrase evaluation sequence. In fact it skips the whole first part of the primitive instruction sequence. After the instruction has been executed the SPECIFY instruction enters the Final Control Sequence at the same point that the imprimitive instructions do, i.e. after the incrementation of the ICT.

The instruction itself consists simply of examining each phrase in sequence and performing the phrase sequence operations. Next the ICT is incremented. At this point the OS is initialized if necessary since the following phrase may need data from the OS. Then the ICT is checked to see if the IBR is empty after which, if the continue bit is on, the next phrase is processed. If the continue bit is off, a check is made for conditional subtraction failure and the CSF and PR#0 are adjusted accordingly.

Note that the SBR value field is used to save all modifications of PR#0 while PR#0 itself is incremented "through" all the phrases. Thus even if there is a change in PR#0 due to one of the SPECIFY phrases, the scan will continue to process all subsequent phrases in the instruction and the modification will only take place at the end of the instruction. The modifications on PR#0 will always be done using the address of the first byte of the instruction as the initial contents of PR#0.

Specify Instruction
SPECIFY Sequence Flow Chart

Section 5.1.8.1

5.1.8.2 Location Instruction

5.1.8.3.2.1 Location Sequence Description

The TP Utility Sequences comprise three instructions LOC, SPECIFY and NOP, all of which are very straightforward. LOC is used to determine the actual 24 bit machine address of a given operand phrase after all of the premodifications have been performed. If the Partitioned Data Access Mode is being used, the final machine address is the one which is obtained.

The LOC sequence works by first setting up the Operand Stack for storing the resultant word of data. Next a memory access is made using the operand address. Note that any pre-scan operations on it have already been performed during the instruction fetch sequence. After the address has been constructed by the memory sequence (using the Partitioned Mode page map if necessary) a test is made in that sequence for a LOC instruction. If it is a LOC, a return is made without actually accessing that address in core. The address, which is in the AR in the memory sequence, is then gated to the LR during which time the leftmost byte is inhibited to zeros. This word is then stored in the top of the Operand Stack and the sequence is finished.

It should be noted that any pre-or post-operations which are performed on the PR designated by the operand phrase tag, remain after the instruction has been completed.

The flowcharts for LOC are given at the end of this section.

The Control Logic for LOC is quite simple. It should be noted, however, that the bits in the Operator Phrase which are normally used as the field designator must be set to '10' for the LOC mnemonic code so that the proper cell size (i.e. word) will be used in ØS ENTRY and ØS WRITE.

Location Instruction
LOC Sequence Flow Chart

5.1.8.4 LOC Control Logic

The LOC control logic is extremely simple since it consists almost entirely of calls to other sequences.

Location Instruction
LOC Control Step Flow Chart

5.2 AU - Executed Primitive Instructions

The Arithmetic Operations cover the following instructions: ADD, SUB, MPY, DIV, POLY, CPRA, CVD, CVF and CVL. Of these, the fixed point versions of CPRA and CVL have been explained under the A/L Compare Sequences in Section 5.1.7 and the Algebraic Unary Instructions in Section 5.1.5.2, respectively. The floating point and decimal versions involve calls to the main AU ENTRY sequence as explained in those sections.

The ADD and SUB fixed point versions are also performed in the TP without recourse to the AU. This will be further shown in Sections 5.2.2.1 and 5.2.2.2. All of the other instructions must use the AU for their execution.

The basic operation of the Arithmetic instructions executed by the AU is performed by having individual initializing sequences for each instruction which set up the control flip-flops for the particular type of operations desired. Then the AU ENTRY sequence is used to obtain the data and make contact with the AU to deliver the instruction and its operands. The final stage makes use of the AU RETURN sequence which obtains the result from the AU and stores it in the proper place.

5.2.1 Basic AU Sequences

5.2.1.1 Basic AU Sequence Descriptions

The basic AU sequences consist of AU ENTRY which obtains operands and sends them to the AU and AU RESULT which takes care of the result coming back from the AU. The first approximation to these sequences will be described in this section.

AU ENTRY begins by checking the OS to make sure that the needed data is present in the hardware registers. Next, it reads out the top cell of the stack. If the instruction requires two fixed point cells, they will both be read out on the same OS READ access by doubling the cell size input to that sequence. In the case of short fixed point numbers, the two operands will emerge in the LR packed in 1 word which must then be separated into two short fixed point operands. This is done by simultaneously using the merge circuitry of the IR and LR and performing permutes and inhibits according to the state of the sign bit as shown in the flow chart at the end of this section.

A single short fixed operand is extended to long fixed by using only the LR and DR merge circuitry.

If the instruction is binary and the number type is floating or decimal, then only the first operand will have been read out of the OS into the LR and DR. In this case, the contents of the LR and DR are stored in the SBR and AR respectively and a second access to the OS is made. Note that WAIT = 1 for the second access, since we will be loading the OS with the result later on, and therefore, there is no sense in changing PR#0 in the case of wraparound since it would only have to be changed back.

Finally, the necessary control byte is generated, and the operands are sent to the AU. Figure 5.2.1.1/1 shows the state of the various registers at this point depending on whether the instruction is unary or binary and depending on its number type. Figure 5.2.1.1/2 shows the order of transmission of the operands for the various types of data.

Note that in sending data to the AU, the second operand in the stack is sent in the first word and the first operand is sent in the second word. In the case of double word operands (i.e. floating or decimal) the leftmost half of the second operand is sent in the first word of data to the AU and the leftmost half of the first operand is sent in the second word. Then the right halves of the second and first operands, respectively, are sent.

The AU ENTRY sequence ends by sending control to the AU RETURN sequence.

The AU RESULT sequence hangs up until the AU has returned the first word of the result. This word is stored in the LR and if a second word is returned it is stored in the DR. Next, a check is made for a bogus result, and if it exists, all the bogus condition flags are loaded into their respective status registers, i.e. OV, FM, UN, LS and ID. Finally, the result is written into the operand stack. The AU will always send the first word to be placed into the stack first and the second word next. If the instruction is decimal DIV, a second double word must be sent by the AU and loaded into the OS.

OS:

Binary Operations:

	SBR	AR	LR	DR
FLT:	2a	2b	1a	1b
BCD:	2a	2b	1a	1b
LFV:	x	x	1	2
SFX:	x	x	1	2

Unary Operations:

	SBR	AR	LR	DR
FLT:	x	x	1a	1b
BCD:	x	x	1a	1b
LFV:	x	x	1	x
SFX:	x	x	1	x

Figure 5.2.1.1/1 Storage of Data before Transmission to AU

Binary Operation
Double-word Operands

Order of Transmission: A₁, B₁, A₂, B₂

Unary Operation
Double-word Operand

Order of Transmission: A₁, A₂

Binary Operation
Single-word Operands

Order of Transmission: A₁, B₁

Unary Operation
Single-word Operand

Order of Transmission: A₁

= 1 word = 4 bytes = 32 bits + 4 flags

ØSP = Operand Stack Pointer

Figure 5.2.1.1/2 - Order of Word Transmission to AU

AU Basic Sequences
AU ENTRY Sequence Flow Chart

AU Basic Sequences
AU ENTRY Sequence Flow Chart

AU Basic Sequences
AU RESULT Sequence Flow Chart

5.2.1.2 AU ENTRY Control Logic

The AU ENTRY control logic uses the control signal UNOP to indicate that the present instruction is a unary operator. It is obtained directly from the instruction code and need not be set up by the calling sequence. Before entry into the sequence, however, the necessary control flip-flops must be set for the OS ENTRY and OS READ control sequences. This includes setting POP to 1 and STOR, to 0. CS2C is to be set according to the number of operands desired from the OS: 0 for one operand, 1 for two operands.

The logic itself is fairly straightforward. In control point AOT2 the cell size is doubled if the number type is fixed and the instruction is unary. In AOT3 the cell size selector must be returned to its original state before the following decision logic will operate.

In control point AOT10 zero, one, or two merges may be performed simultaneously, depending on the state of the sign bits, IR19 and LR19. Note that these bits will not change during the merge so that no interruption of the task will occur.

AU Basic Sequence
AU ENTRY Control Step Flow Chart

AU Basic Sequence
AU ENTRY Control Step Flow Chart

5.2.1.3 AU RESULT Control Logic

Although the AU RESULT control logic contains only eight control points, it is moderately complicated because of the various decisions which have to be made about which operations to perform. There are several control signals which are needed for the sequence operation. CVL, CPRA, MPY, DIV and POLY all come from the basic instruction decoding logic. The bogus result signal is currently the sixth bit in the control byte coming from the AU. There are no direct control signals which need to be set before entering the sequence.

The sequence does not actually start until a return signal is given by the AU signifying that an operand is waiting on the data bits of the OUTBUS from the Exchange Net. After the one or two words of data have been transmitted a set of decision logic is entered which determines which control point is entered next. The possibilities are AOT33, AOT34, AOT37, or AOT38. Figure 5.2.1.3/1 lists the boolean expressions for each in terms of the advance out signals of the control points and the various control lines.

Control point AOT38 is pretty complicated since there are so many possible variants for its task. The necessary logic for its task output signal is shown in Figure 5.2.1.3/2.

	$(AOA031 \cdot CVL \vee AOA032) \cdot (\overline{\text{Bogus Res}} \vee CPRA)$	}	33
	$(AOA031 \cdot CVL \vee A)A)32) \cdot (\overline{\text{Bogus Res}} \vee CPRA) \cdot (SFX \cdot DIV)$	}	34
v	$AOA033 \cdot \overline{CPRA} \cdot (SFX \cdot DIV)$	}	
	$(AOA031 \cdot CVL \vee AOA032) \cdot (\overline{\text{Bogus Res}} \vee CPRA) \cdot (SFX \cdot MPY)$	}	37
v	$AOA033 \cdot \overline{CPRA} \cdot (SFX \cdot MPY)$	}	
	$(AOA031 \cdot CVL \vee AUA032) \cdot (\overline{\text{Bogus Res}} \vee CPRA) \cdot \overline{SFX}$	}	38
v	$AOA033 \cdot \overline{CPRA} \cdot SFX$	}	
v	$AOA036$	}	
v	$AOA037$	}	

Figure 5.2.1.3/1

Boolean Equations for AU RESULT Decision Logic

Figure 5.2.1.3/2 - Task Signal Logic for AOT38

Basic AU Sequences
AU RESULT Control Step Flow Chart

Section 5.2.1.3

5.2.2 Arithmetic Instruction Sequences

The Arithmetic Instructions may be performed either in the AU or the TP, depending on the instruction and the number type of its operands. The following sections describe the operation of ADD, SUB, MPY, DIV, CVD, CVF, and POLY. The operation of CPRA is described in Section 4.3.2.2 and CVL is described in Section 4.3.3.2. The ABS, NEG, and MNS instructions, which also might be considered arithmetic, are also described in this latter section.

5.2.2.1 Arithmetic Instruction Sequence Descriptions

The purpose of the arithmetic instruction sequences is to perform those arithmetic instructions which can be done in the TP and to prepare the initial conditions for entering the AU ENTRY sequence for those instructions executed in the AU.

Figure 5.2.2.1/1 contains a table listing the arithmetic instructions which can be performed in the AU and indicating the number of operands sent to and received from the AU across the Exchange Net. It also lists the indicators which may be effected by each instruction and the actual number of words sent to the TP from the AU completion of the instruction. Figure 5.2.2.1/2 shows the position of the various indicator flags in the data words sent by the AU. These flags are all contained in the flag positions of the data bytes. Note that the AU never returns a short fixed operand. This is because the TP always extends the sign bit on short fixed operands before sending them to the AU. The result then returns to the TP as a single long fixed point operand (in MPY) or two long fixed operands (in DIV). In the DIV instruction only the right half of each operand is stored in the stack.

In the ADD and SUB instructions the first step is to check the OS to make sure it contains two operands. In the case of a floating or decimal number type this is done by transferring control directly to AU ENTRY A and proceeding from there. If the number is fixed point OS ENTRY is executed and then OS READ is used to obtain the topmost operand from the OS. If the instruction is a SUB the operand must be obtained in 1's complement form. The stack pointer is decremented the length of the operand and if wraparound occurs, PR#0 is automatically decremented.

After OS READ the data in the LR is stored in the DR before a second access is made. Since the data must be right-justified, short fixed operands must be permuted.

		No. Operands	No. Ops. returned to Stack	Indicators	No. Words transmitted From AU
ADD	FX	2	1		TP
	FLT	2	1	OV, UN, LS	2
	BCD	2	1	OV, ID	2
SUB	FX	2	1		TP
	FLT	2	1	OV, UN, LS	2
	BCD	2	1	OV, ID	2
MPY	FX	2	2	OV	2
	FLT-BCD	2	1	OV, UN, OV, ID	2
DIV	FX	2	2	OV	2
	FLT	2	1	OV, UN	2
*	BCD	2	2	OV, ID	4
CPRA	FX	2	0	GT, LT, EQ, FM	TP
	FLT	2	0	GT, LT, EQ, FM	2
	BCD	2	0	GT, LT, EQ, FM, ID	2
CVD	FX	1	0		2
	FLT	1	0	OV, LS	2
CVF	FX	1	0		2
	BCD	1	0	ID	2
CVL	FX	1	0		TP
	FLT	1	0	OV, LS	2
	BCD	1	0	OV, LS, ID	2
POLY	FX	n	2	OV	2
	FLT	n	1	OV, UN, LS	2
	BCD	n	1	OV, LS, ID	2

Figure 5.2.2.1/1 - Arithmetic Instructions Performed by AU

On the second read from the OS, the OS pointer is also decremented. However since the DR contains data which must not be destroyed and since the OS pointer will be moved back before the end of the instruction, WAIT is set to 1 so that PR#0 will not be changed in the case of OS wraparound.

After the second operand has been read the addition is performed. A CINJ must be generated if the instruction is a SUB. The result from the AR is then loaded into the LR to be written back into the OS by the OS WRITE sequence. Note that once again short fixed numbers must be permuted so that the operand is left-justified in the LR.

Finally the state of the adder overflow flip-flop (AOV) is saved in the OV status register. It should be noted that the adder logic looks at the proper bit position depending on the cell size to determine the overflow conditions.

The MPY and DIV instructions are quite simple. In the case of a MPY with fixed point operands or a DIV with either fixed point or BCD operands, two operands will be returned. In the case of a fixed point MPY the result consists of high and low order parts of the result and in fixed point and BCD DIV, the result consists of a quotient and remainder. In either case the WAIT signal for the first OS READ in the AU ENTRY sequence must be turned on since we will be reloading the OS to the same point as before the execution of the instruction and PR#0 should not be modified in the case of OS wraparound.

CVF and CVD are extremely simple since they consist of entering the AU ENTRY sequence at entry point B.

The POLY instruction does not make use of the AU ENTRY sequence since POLY is considerably different from the other arithmetic instructions. The OS format for POLY is shown in Figure 5.2.2.1/3. The first step is to check the OS to make sure it has a halfword in the hardware registers. Then this count is accessed and stored in the SBR value field.

* = Not Used in this Number Type

☒ = Flag of Byte (Bit #9)

Figure 5.2.2.1/2 Flag Bit Designation
for Arithmetic Indicators

$$\text{Result} = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

Figure 5.2.2.1/3 - OS Format for POLY Instruction

At this point the POLY loop is entered and the OS hardware is accessed to get the X operand. Then the SBR value field is checked for zero. If it is zero the instruction is completed. Since the polynomial is zero order the answer equals the first coefficient (which is at the top of the stack at this point) and there is no need to access the AU. If it is not zero the x operand is sent to an AU. Note that on the first time through the POLY loop, the count is not decremented since a coefficient is not being sent. On each succeeding loop, however, the count is decremented. Eventually, when the count reaches zero, a QUIT signal is sent to the AU indicating that there are no more operands. Then control is passed to the AU RESULT sequence to process the data returned by the AU.

Arithmetic Instruction Sequences
ADD and SUB Sequence Flow Chart

Arithmetic Instruction Sequences
MPY, DIV, Sequence Flow Charts

Arithmetic Instruction Sequences
POLY Sequence Flow Chart

5.2.2.2 ADD and SUB Control Logic

The ADD and SUB instruction control logic is very simple. Both instructions use the same sequence except that SUB sets the NEG control flip-flop to 1 for the first OS READ sequence and ADD does not.

Note that the floating point and decimal versions of the instruction are performed under control of the AU ENTRY sequence.

Arithmetic Instruction Sequences
ADD and SUB Control Step Flow Chart

5.2.2.3 MPY and DIV Control Logic

The MPY and DIV control sequences are very short and simple. They share control points AOT41 and AOT49 with the ADD and SUB control logic.

Note that if the instruction is fixed point, the WAIT signal must be turned on for the first OS READ sequence execution.

Arithmetic Instruction Sequences

MPY ENTRY and DIV Entry Control Step Flow Chart

5.2.2.4 POLY Control Logic

The POLY control logic contains a loop which slightly complicates the sequence. The loop consists of control points AOT56 through AOT66. Note that it is executed differently on the first time through. The first pass is detected by the fact that after control point AOT59, the advance out signal from AOT66 will be off only for the first pass. This is due to the fact that it is set by the general set mechanism at the beginning of the sequence. On succeeding loops however it is not set until the execution of AOT62 which will not occur until after the decision has been made.

Note the POLY is only valid for floating point operands. All other number types cause an illegal instruction interrupt.

Arithmetic Instruction Sequences

POLY Control Step Flow Chart

Section 5.2.2.4

5.3 PAU - Executed Primitive Instructions

There are 36 instruction variants which can be sent to the Pattern Articulation Unit (PAU) from the Taxicrinic Processor. The purpose of this section is to describe the operations which the TP must perform on the instructions and data before they are sent to the PAU.

There are several possible actions which may be performed during the processing of a PAU instruction by the TP. The instruction mnemonic byte itself is always sent to the PAU without any modifications. In addition there may be other bytes which are also sent to the PAU directly from the instruction string.

In some instructions there is a count phrase. This may be long or short. There may also be one to three plane address phrases. These may be long or short and indicate which PR's contain the plane names to be used by the PAU. There is sometimes one final phrase which indicates the location of a data field which is to be used by the PAU. For all of these phrases the TP must perform the indicated phrase operations and send the proper data to the PAU via the Exchange Net.

Finally, if the PAU replies after receiving an instruction, the TP must accept the reply and save it in an appropriate location for the programmer.

5.3.1 PAU Instruction Formats

The purpose of this section is to describe the format of the PAU instructions as they are received by the TP in order that the PAU instruction sequence can be more easily understood. The difference between PAU instructions and other primitive and imprimitive instructions will also be described.

As was mentioned in the last section each PAU instruction begins with a mnemonic byte. In addition to the mnemonic byte there may be one or more data bytes which contain various types of information which will be needed by the PAU to execute the instruction, e.g. GBW byte, identification byte, etc.

After one or two bytes which are used directly by the PAU may come one or more operand phrases. Unlike the primitive instructions, there is no need to limit the total length of all of the phrases. This is due to the fact that the TP only refers to the information once - when it sends it to the PAU. Thus there is no need to save the phrase for possible use later on in the execution of the instruction.

In line with the above, the TP scans the operand phrases of a PAU instruction, if they are present, and processes each one completely before proceeding to the next. In turn it performs the pre-operations, sends off the required data to the PAU, and then does the post-operations if necessary.

There are three possible kinds of phrases which may be processed: count phrases, plane address phrases, and normal memory access phrases. The count phrases are essentially the same as those described for the multi-cycle instructions in Section 5.1.2 except that they may be either long or short. The two byte count is taken from the value field of the indicated PR after the specified modifications have been performed and is set to the PAU as part of the data stream. The count is not optional and must be set to 0 by the assembler, if not used.

The plane address phrases are somewhat similar in that the final address that is sent to the PAU is actually taken from the pointer register. The plane address phrases may be long if desired. In many of the instructions more than one plane address is present. In these cases each plane address phrase except the last has its continue bit turned on. In this respect the PAU instructions are like the imprimitives. The last plane address byte sent to the PAU during an instruction has its flag turned on.

The so-called memory access phrases are exactly the same as a normal operand phrase in a primitive instruction. Each PAU instruction has at most one, and if it is present it occurs at the end of the instruction. It is used either to access a data field in core or, as in the SHIFT, TALLY, TALLYHO and CONNECT instructions, to access a one byte displacement or direction list.

Section 5.3.2 gives a detailed description of the PAU MAIN instruction sequence. The basic idea is to use the mnemonic register to set a series of control signals which indicate which of the various substructures mentioned in this section may be present. These signals are then used to process the instruction, one phrase at a time.

TWOB: indicates that there are two bytes in the instruction which are sent directly to the PAU.

COUNT: indicates that there is one count phrase.

CONTP: indicates that there are one or more plane address phrases.

FINF: indicates that there is a memory access operand phrase which accesses a field.

FINB: indicates that there is a memory access operand phrase which accesses a byte.

RASTER: indicates that the instruction needs a raster string.

WAT: indicates that the instruction will cause the PAU to send back data to the TP.

OSD1: indicates that 4 bytes of data must be sent to the PAU from the OS.

OSD2: indicates that 6 bytes of data must be sent to the PAU from the OS.

W1, W2: used as an index to indicate 4 possible situations when waiting for data to return from the PAU.

00: 2-byte count returned.

01: unknown length string of bytes.

10: EQ flip-flop setting.

11: EQ setting plus 1 byte of data.

The use of these signals will be explained in greater detail in Section 5.3.2.

	TWOB	CONF	COUNT	FINF	FINB	OSD1	OSD2	TBYT	RASTER	WAT	W1	W1
BOOLE	1	0	0	1	0	0	0	0	0	0	-	-
AREA	0	1	1	0	0	0	0	0	0	1	0	0
LIST	0	1	1	0	0	0	0	0	0	1	0	1
LISTSZ	0	1	1	0	0	0	0	0	0	1	0	1
LISTLZ	0	1	1	0	0	0	0	0	0	1	0	1
READLZ	0	0	0	0	0	1	0	0	0	1	-	-
RDERLZ	0	1	0	0	0	1	0	0	0	1	-	-
WRITLZ	0	1	0	0	0	0	1	0	0	0	-	-
WRERLZ	0	1	0	0	0	0	1	0	0	0	-	-
TESTP	0	1	0	0	0	0	0	1	0	1	1	0
TESTB	0	1	0	0	0	0	0	1	0	1	1	1
LISTI	0	1	0	0	0	0	0	1	0	1	0	1
ERASEP	0	1	0	0	0	0	0	1	0	0	-	-
SHIFT	0	1	0	0	1	0	0	0	0	0	-	-
TALLY	0	1	0	0	1	0	0	0	0	0	-	-
TALLYHO	0	1	0	0	1	0	0	0	0	0	-	-
CONNECT	0	1	0	0	1	0	0	0	0	0	-	-
LOADB	1	0	0	1	0	0	0	0	1	0	-	-
PUSHB	1	0	0	1	0	0	0	0	1	0	-	-
STORB	1	0	0	0	0	0	0	1	0	1	0	1
POPB	1	0	0	0	0	0	0	1	0	1	0	1
CLEARP	1	1	0	0	0	0	0	0	0	0	-	-
SETP	1	1	0	0	0	0	0	0	0	0	-	-
MOVEB	1	1	0	0	0	0	0	0	0	0	-	-
PLOT	0	1	0	1	0	0	0	0	0	0	-	-
PLOTSZ	0	1	0	1	0	0	0	0	0	0	-	-
PLOTLZ	0	1	0	1	0	0	0	0	0	0	-	-
PLOTI	0	1	0	1	0	0	0	0	0	0	-	-
COPY	0	1	0	0	0	0	0	0	0	0	-	-
COPYC	0	1	0	0	0	0	0	0	0	0	-	-
TOPOLOGY*	0	0	0	0	0	0	0	0	0	0	-	-
REPLICATE	0	1	0	0	0	0	0	0	0	0	-	-
GATEIA	0	0	0	0	0	0	0	0	0	0	-	-

Figure 5.3.1/1 - Control Flip-Flop Settings for PAU Instruction

*Four Instruction Variants

5.3.2 PAU Main Sequence

5.3.2.1 PAU Main Sequence Description

After the instruction fetch sequence has determined that the present instruction is a PAU instruction, control is given to the PAU Main sequence. The PAU flowcharts are given at the end of this section.

The PAU sequence first decodes the last 6 bits (exclusive of the flag bit) in the instruction byte to set the control flip-flops. These flip-flops are then used to process the instruction sequentially one phrase at a time.

Once the flip flops are set, the processing begins. If the instruction is a GATEIA, a special sequence is used which obtains the IA word from the IBR if the IA word is entirely in it or by a PR#0 memory access if it is not. Then the sequence accesses the PAU, sends the mnemonic byte and the IA word and then goes to the PAU RESULT sequence.

If the instruction is not a GATEIA instruction and if there are two bytes in the instruction which are to be transferred to the PAU directly, the IBR is checked to see if it contains both bytes. If not, it is reloaded and the IR is filled with the new data, being careful not to destroy the good data that is already there. Then the ICT is incremented by 2.

If there is only one byte to be sent directly to the PAU unchanged, there is no need to check the IBR since at the beginning of an instruction sequence the IBR always contains at least one good instruction byte. In this case the ICT is only incremented by one.

In either case the data is gated into the SBR. At the same time it is permuted left zero or one byte so that the bytes to be eventually sent to the PAU are in the SBR right justified. The remaining bytes of the SBR are inhibited to zero.

If the CONTP flip flop is on, the plane address operand decoding section is used. This section loops until it completes a plane address phrase which has its continue bit off. Note that during the phrase processing there can be no PR#0-SBR exchange since the SBR contains special information. For this reason, no PAU instruction operand phrase may have a tag of zero. After the phrase processing, the LR is loaded with the right-most byte of the PR specified by the tag field of the phrase. This is then merged with the data in the SBR by loading the DR from the SBR and performing a DR → LR merge. Finally the LR is stored back into the SBR and the post operation is performed.

After the plane address sequence detects a continue bit which is off the count flip-flop is checked. The count phrase is processed in the same manner as the plane address phrase except that two bytes are needed from the PR. After processing there will be no further data since the count is the last field of those PAU instructions which have a count. Finally access is made to the PAU to transfer one word of instruction plus data and control is sent to the PAU RESULT sequence.

If there is no count, the next step is to check for a final operand phrase (FINB or FINF = 1). If there is one it is processed and then if the operand indicates a field (FINF = 1) control is sent to the PAU FIELD ACCESS sequence. If the operand indicates a byte, a one byte read access is made to memory. If the instruction is CONNECT, two words must then be sent to the PAU while if it is not only one is sent. Finally the post operations are performed on the phrase and control is given to the PAU RESULT sequence.

If there is no final operand phrase OSD1 and OSD2 are checked. If either of these are on data must be obtained from the OS. If OSD2 is on 6 bytes will be popped. If OSD1 is on 4 bytes will be popped and later on 6 bytes will be pushed. In either case 2 words of data are sent to the PAU before control is transferred to the PAU Sequence.

If no data is to be taken from the PAU then there are no other operations to be performed and the data in the SBR is sent to the PAU.

The interrupts in the PAU Main sequence follow the same general pattern shown in the previous control sequences, except that nothing is ever undone. If an interrupt is detected in any sequence called by the Main PAU sequence, control is given to the interrupt sequence. When the interrupt processing returns, control will be given to the immediately previous interrupt return point in the sequence.

PAU Main Sequence Flow Chart

PAU Main Sequence Flow Chart

Section 5.3.2.1

p. 3/4

PAU Main SEQUENCE FLOW CHART

5.3.3 PAU Utility Sequences

5.3.3.1 PAU Utility Sequence Descriptions

The PAU utility sequences consist of the PAU ACCESS Sequence and the PAU FIELD ACCESS Sequence. The PAU ACCESS Sequence is used to control the transmission of data from the TP through the Exchange Net to the PAU. The sequence has the option of ending the access after sending one or two words or of informing the PAU that more data will be forthcoming. In the latter case the connection with the PAU is broken but only temporarily.

The PAU FIELD ACCESS Sequence, as mentioned in Section 5.3.2.1, is used in the final stage of processing of those PAU instructions which use a field of data which is located in core. These include the plot instructions: PLOT, PLOTSZ, PLOT LZ, and PLOT I, the raster field instructions: LDBORD and PUSHROW, and the BOOLE instruction. The sequence itself is divided into two sections, one which handles the raster instructions and the other which takes care of the BOOLE and plot instructions.

The sequence begins by getting rid of the data which has already been collected by the Main PAU sequence and stored in the SBR. The BOOLE and Plot instruction part of the sequence takes this data, which consists of one byte of mnemonic and either an availability list byte for BOOLE or a plane address byte for the plot instructions, and sends it to the PAU.

Next a double word is accessed from the field in core. Various checks are made on this data to see if the end of the field has occurred. These checks depend on the instruction in the following manner:

- 1) If the instruction is a PLOT, PLOTSZ, or PLOT LZ the end of the field is signified by 2 flags in a row being 1. The two bytes with the flags do not contain data but must be sent to the PAU since the PAU control also uses the flags to detect the end of a field. The control signals for the PAU access are set according to where the "on" flags are detected.
- 2) If the instruction is a PLOT I, the field is ended by any byte which contains an incremental command. The control signals to the PAU Access Sequence are set so that the word containing the command is sent to the PAU.

- 3) If the instruction is a BOOLE the field is ended by a single byte with a flag of 1. This byte is the last one which must be sent to the PAU.

After the check for the end has been made, the PR controlling the scan of the field must be incremented. If the end of the field was found in the first word the PR is only incremented by four. If the end was found in the second word, or if no end was found, the PR is incremented by 8.

Next the PAU Access is made and if the end was not found in the last data the cycle is repeated. Otherwise control is given to the PAU RESULT sequence.

It should be noted here that since no attempt is made by the hardware to access at double word boundaries, it is to the programmer's advantage, as far as execution time is concerned, if the assembler locates his fields so that the data begins on a double word boundary. In this manner it will not take two separate memory accesses to get the double word.

The Raster Instructions have three distinct phases. In the first phase the data already obtained by the Main PAU sequence is sent to the PAU. Before this is done, however, the GBW byte is checked to see if the left border is to be included in the data. If so, the PR must be decreased by one and a byte access must be made. This byte is then sent to the PAU in the rightmost byte position while the other two bytes are in the leftmost positions.

The second phase involves sending the data for the main 32 positions of the raster strung. If only 1 bit of grey scale is used there will only be one word of data. If 2, 4, or 8 bits of grey scale are used, there will be 1, 2, or 4 double words of data. The Raster counter RSCT is loaded with the correct number of double words for these three cases by decoding the grey scale level bits in the GBW byte.

Then the proper number of double words are read from core and sent to the PAU.

The final phase is performed after the data for the main 32 positions of the raster string have been sent. It consists of checking to see if the right border is to be sent to the PAU. Although the GBW byte is no longer accessible in this phase, the FBYT flip-flop was set to 1 during phase one, if the right border was to be sent.

After the check and possible data access and transmission the control goes to the PAU RESULT sequence.

QUIT = 1 send QUIT signal to PAU
 Q2ND = 1 if QUIT=1, send QUIT on second word of access
 if QUIT=0, send two words, if Q2ND=0, QUIT=0 and one word

PAU Utility Sequence
 PAU ACCESS Sequence Flow Chart

PAU Utility Sequences
PAU FIELD ACCESS Control Sequence Flow Chart

PAU Utility Sequence
PAU FIELD ACCESS Sequence Flow Chart

5.3.4 PAU FINI Sequence

5.3.4.1 PAU FINI Sequence Description

The PAU FINI Sequence is used to finish up the PAU instruction sequences. In the case where data will be returned, this means that it must obtain the data and store it in the correct place.

The first step is to check the WAT control flip flop to see if any data will be returned by the instruction. If there will be no return, control is given to the Final Control Sequence of the imprimitive instructions at entry point B.

If WAT = 1 then the sequence sits and waits for a signal from the PAU that some data is on the lines from the XN. While it waits, the sequence checks for interrupts and if one occurs, it notifies the PAU and then transfers to the interrupt sequence.

When the PAU finally returns some data it is stored in the LR. If the OSD1 control flip flop is one, a total of 6 bytes of data will be returned to be pushed into the stack. Therefore a second word will be sent by the PAU and the TP will store this word in the DR. Then a double word OS WRITE is performed. This will not cause any overflow problems even though there are only 6 bytes of data since in the MAIN PAU sequence we made allowances for pushing in a double word in this case. However after the write is completed the SCR must be decreased by 2 bytes to indicate the proper top to the OS.

If the OSD1 flip flop is not on, then W1 and W2 are used as a 2 bit register to indicate which of 4 possible cases may occur. These flip flops were originally set when the op code of the instruction was first decoded.

If W1 and W2 are zero, the instruction is AREA and two bytes are returned to the OS.

If W1 is a '1' then the instruction is either TESTP or TESTB. In these cases the EQ status flip flop is set according to the state of

the second flag from the left in the LR. If W2 is a '1' the leftmost byte is also pushed into the OS.

If W1 and W2 are '01' then an unspecified number of bytes will be sent from the PAU to the OS.

PAU FINI Sequence Flow Chart

5.5 Imprimitive Instructions

The imprimitive instructions include GO TO, CALL, EXECUTE and EXIT. Their main purpose is to effect transfers of control from one instruction sequence to another.

In general the imprimitives consists of a mnemonic byte followed by one or more long or short operand phrases.* The structure of the zeroth operand phrase (henceforth referred to as the operator phrase) is similar to the standard operand phrase format described in the Programming Manual except that the slashing bits are ignored. No pre-or post-slashing is allowed on the zeroth operand phrase. The pre-operation modifications are allowed however.

The remaining operand phrases may be long or short and may specify any of the pre- operations as well as pre- and post-slashing. However, since the GO TO and EXIT instructions never have a post-scan sequence, the post-slashing in these instructions will never be performed.

It should be noted that if the special pointer registers, PR#0 or PR#13 are modified, the hardware will automatically make sure that the Instruction Buffer Register (in the case of PR#0) and/or the hardware storage for the Operand Stack (in the case of PR#13) are cleared and/or initialized at the appropriate times.

If a conditional subtraction fails at any time during the pre-scan of an imprimitive instruction, that particular modification is not performed. The scan continues to the end however at which time the conditional subtraction failure actions for that particular instruction are performed.

In the following sections detailed descriptions of the individual imprimitive instructions are given. These are followed by a description of the pointer stack manipulations which are performed by each instruction. Then the name permutation sequence is introduced and finally the flowchart for the full sequence is described.

*An exception is the EXIT instruction which consists solely of a mnemonic byte.

5.5.1 Imprimitive Instruction Descriptions

5.5.1.1 GO TO

The GO TO instruction is used to unconditionally transfer control to another segment or to another part of the same segment. It consists of a mnemonic byte followed by one or more long or short operand phrases.

The zeroth operand phrase tag indicates which pointer stack will become the new PR#0 at the end of the GO TO instruction. If it is zero no name change will occur. Otherwise the name of the pointer stack mentioned by the operator tag and the name of the current PR#0 stack will be exchanged. The pre-modification indicated in the zeroth operand phrase will be performed on the PR named by the tag. If the tag is zero the modification will take place on the initial value of the PR, i.e. its value before the scan began, but will not become effective until the end of the scan.

There may be any number of operand phrases following the operator phrase and any PR name may be used in an operand tag as often as desired. Whenever a PR is used more than once the second modification is performed on the results of the previous modifications to that PR and its stack. If PR#0 is mentioned the results will not affect the scan until the end, and then only provided no conditional subtraction failure has occurred. If a conditional subtraction failure does occur at the end of the scan, the top of PR#0 will contain the address of the next consecutive instruction instead of a modified value. However if any pre-slashing was performed, the values which were pushed at the time of the pre-slashing will still be left in the pointer stacks.

At the end of the scan of the GO TO, provided no conditional subtraction failure has occurred, control will be given to the instruction whose address is in the top of the new PR#0 stack. If conditional subtraction failure did occur there will be no exchange of control pointer names; control will therefore continue to reside with the present PR#0 - which will now point to the next instruction in sequence immediately following the GO TO.

There are no modifications of stack depth intrinsic to the GO TO instruction itself although pre-slashing can be used to increase stack depth where desired. Since there is no post-scan of a GO TO instruction, the GO TO cannot decrease stack depth.

5.5.1.2 CALL

The CALL instruction is used to transfer to a procedure sequence. It is capable of linking actual with formal parameters by permuting the names of the various pointer register stacks. A CALL consists of a mnemonic byte followed by one or more operand phrases.

The 0th operand phrase tag indicates the PR which will contain the procedure address at the end of the pre-scan. This stack will become the new PR#0 stack and its' top three entries will be:

First:	Address of the procedure
Second:	Return Address
Third:	Old top of stack before CALL began

The operand phrases may be long or short. Each PR which is mentioned in an operand phrase will be renamed at the end of the scan according to its position in the scan, i.e. the 0th operand phrase indicates which will be renamed 0, the first operand phrase indicates which will be renamed 1, etc. If it happens that a PR is mentioned by more than one operand phrase, each subsequent mention will be ignored. These latter modifications will not be performed and the scan will continue as if they did not exist.

The pre-operations will be performed on each phrase in turn. At the end of the scan if no conditional subtraction failure has occurred the new PR#0 stack will be set up as described above and control will be sent to the new PR#0, i.e., to the procedure sequence.

If there was a conditional subtraction failure, a post-scan of the CALL will be performed as soon as the pre-scan is completed. The stack which was to have been the new PR#0 will be restored to its original depth (not counting the effects of pre- or post-slashes). The top of PR#0 will point to the next instruction; accordingly any modifications to PR#0 will be cancelled with the exception of anything which may have been pushed into the PR#0 stack by a pre-slash and not removed by a post-slash.

At the end of the post-scan, control is given to the next instruction in sequence after the CALL.

If the CALL is successful, there will eventually be a return caused by an EXIT instruction. When this occurs the post-scan of the CALL will be performed as a part of the EXIT sequence. The EXIT sequence will cause the proper number of items to be popped from the stacks so that the levels before the CALL was begun will be restored (see Section 5.5.2.3).

5.5.1.3 EXECUTE

The EXECUTE instruction has the same format and effect as the CALL instruction except that when it is executed the execute flag in the new PR#0 is set to "1". This flag is checked at the end of every instruction to determine if the instruction just completed was called by an EXECUTE. If it was, an automatic EXIT instruction is performed.

5.5.1.4 EXIT

The EXIT instruction provides a means of returning from a CALL sequence. It consists of one mnemonic byte.

The EXIT instruction will pop the PR#0 stack once, transfer to this return address and check for the presence of a CALL instruction at that point. If the instruction is a CALL, its post-scan is performed. If it is any other instruction, the instruction is executed in the normal fashion.

The stack depth during an EXIT is changed as follows: the PR#0 stack is popped once during the EXIT and again during the post-scan of the CALL at the return address. If there is no CALL at the return address, the level of the stack is reduced by popping out the second entry.

5.5.2 Pointer Stack Manipulation

In transferring control from one instruction sequence to another, the imprimitive instructions must sometimes affect the depth and contents of several of the pointer stacks. This is especially true for the CALL, EXECUTE and EXIT instructions. The purpose of this section is to give a detailed view of the pointer stack manipulations as the imprimitive control sequence is executed. In going through each instruction it will be helpful to follow along on the main imprimitive flowchart at the end of Section 5.5.4.1.

In the stack diagrams the following conventions will be used:

a_i = contents of the top of stack i .

a'_i = contents of the top of stack i after modification.

β = length of imprimitive instruction.

Stacks will be represented as vertical stacks as shown in Figure 5.5.2 where the top of the stack is toward the top. Link fields are not represented. The segment name is represented by a small square to the right of the pointer value and the execute flip-flop is represented by a small square to the left of the pointer value. The status of the Spare Buffer Register will be shown at each stage. All fields of the SBR including the "link" are shown.

Figure 5.5.2 - Example of PR Stack Representation

5.5.2.1 GOTO Stack Manipulation

Figure 5.5.2.1/1 GOTO: Before Prescan

Figure 5.5.2.1/2 GOTO: Following the Prescan

Figure 5.5.2.1/3 GØTØ: Post-execution, no CS failure

Figure 5.5.2.1/4 GØTØ: Post-execution, CS failure

5.5.2.2 CALL and EXECUTE Stack Manipulation

Figure 5.5.2.2/1 EXECUTE (or CALL): Following processing of the operator phrase

Figure 5.5.2.2/2 EXECUTE (or CALL): Following the Prescan

Figure 5.5.2.2/3 EXECUTE (or CALL): On Entry to the Postscan

Figure 5.5.2.2/4 EXECUTE (or CALL): Following Completion of the Postscan

5.5.3 Pointer Name Permutation

Name permutation is a process which must be performed on the Name Registers of the Pointer Registers whenever a CALL or EXECUTE instruction is to be performed. These instructions consist of an initial mnemonic byte followed by one or more operand phrases of 1 or 3 bytes.* Each phrase may indicate certain pointer operations which are to be performed on the PR whose name appears in the "tag" portion of the phrase. Basically, these imprimitive operations are changes of control in which the operator phrase indicates the type of change while the remaining operand phrases manipulate various parameters before the change is made (and also after the "return").

Since the section of code to which control is being transferred will want to refer to certain of the parameters (whose addresses are in the PR's), these parameters must be given "standarized" names. This is what is done in the name permuting process. Another way of stating this is to say that the actual parameters of the old control sequence are linked up to the formal parameters of the new control sequence.

The renaming process is performed so that the old PR names are changed to new names, beginning with 0,1,2 and possibly on through 13. Thus the PR indicated by the tag of the beginning operator phrase is renamed 0, the PR named by the tag of the first operand phrase is renamed 1, and so on to a maximum of 13. If any operand phrase tag contains a name which has been mentioned in the tag portion of a previous phrase in the instruction, it will be ignored. None of the modifications specified by the phrases with duplicate tags will be performed.

*See Section 3 of the Illiac III Programming Manual for a complete description of imprimitive instructions.

As an example, the instruction `CALL 1 = A, 1 + B, 2, 0;` will cause PR#1 to be renamed PR#0. The second phrase will be ignored and thus PR#2 will be renamed PR#1 and PR#0 will become PR#2. It should be noted that the contents of the new PR#0 (i.e. old PR#1) will only contain A since the second phrase in the instruction will not modify the PR.

In the above example the CALL instruction specified a complete permutation, i.e. the set of new names and the set of old names are exactly the same with the difference being that the names are attached to different physical pointer registers. However, consider the instruction `CALL 1, 2;`. Here PR#1 is renamed PR#0 and PR#2 is renamed PR#1. But now we have two PR#0's, the old one and the new one which used to be PR#1. Obviously since the names must be unique we must change the old PR#0 name to something else. Intuitively we can see that the only name available is PR#2. In the TP imprimitive sequence the hardware must be able to take care of this type of situation and complete the permutation so that all of the names are unique.

In order to take care of this problem, the imprimitive sequence must check the new names at the end of the scan of the instruction. If the highest new name assigned was *i*, then it must check each old name, less than *i*, to see if it was renamed. If any of them have not been renamed, then they must be given a new name so that there will not be a duplication of names. The new names are taken from those names in the original tag sequence which were greater than *i*.

An example will hopefully demonstrate this process more clearly. Given the tag sequence 11, 8, 3, 5, 0, the name changes will occur as follows:

"old" PR Names	0	1	2	3	4	5	6	7	8	9	10	11	12	13
"new" PR Names	4	5	8	2	11	3			1			0		

- 1) the PR now named 11 will be renamed 0
- 2) the PR now named 8 will be renamed 1
- 3) the PR now named 3 will be renamed 2
- 4) the PR now named 5 will be renamed 3
- 5) the PR now named 0 will be renamed 4

At this point the tag sequence ends so the old PR's named 0 through 4 must be checked to see if they were renamed (the highest new name at this point is 4).

- 6) old PR 0 has been changed -- do nothing.
- 7) old PR 1 has not been changed -- rename it 5.
- 8) old PR 2 has not been changed -- rename it 8.
- 9) old PR 3 has been renamed -- do nothing.
- 10) old PR 4 has not been renamed -- rename it 11.

In the actual control sequencing described in Section 5.5.3.2 the operations performed are slightly different than these but their effect is the same.

If and when the "new" control sequence decides to return to the "old" sequence, the TP must "undo" the name permutation to get back the old PR names. This is done by rescanning the imprimitive instructions and performing the opposite operations from those given above. This process is called "repermutation" and will be explained in more detail in Section 5.5.3.2.

5.5.3.1 Name Permutation Hardware

The name permutation process is performed by the PR control registers with direction from the main TP control. The Pointer Register control registers include 15 Name Registers (NR's), 15 Shadow Name Registers (SNR's), 15 one-bit Name Flag Registers (NFR's), the Tag Register (TGR) and two counters (CCT and ACT). Figure 2.4.2.2 is reproduced in more detail in Figure 5.5.3.1/1 to show the relationships of these registers to the two busses: the Name Bus (NB) and Shadow Name Bus (SNB) of the PR control logic.

As described in Section 2.4.2.2 the Name Registers are used to hold the current name for each physical PR, while the Tag Register stores the tag portion of the operand phrase currently being evaluated.

The Shadow Name Registers are 4-bit registers which can be loaded by gating in from the Shadow Name Bus. Each Shadow Name Register can be gated out to its corresponding Name Register. The Shadow Name Registers are used to store the "new" names for the PR's while the renaming process is in progress. When the name permutation has finished, the SNR's are gated into the NR's, thereby completing the name change.

The Name Flag Registers are 1-bit registers which are assigned to each SNR. (The one bit NFR and 4 bit SNR are implemented using one 247-00 circuit card). During the prescan, whenever a PR is renamed, and the new name is gated to the corresponding SNR and its NFR is turned on. By this means when the tag sequence has been finished, the TP control can tell which of the PR's have been changed. The NFR's are also used when the SNR's are gated into the NR's. As will be explained in section 5.5.3.2 if a PR has not been renamed during the name permutation, the SNR is not gated into the corresponding NR.

The outputs of the NFR's are tested in the following manner. First, all outputs are gated by a common "NFR check" signal, NFCK/E, and dot-or'ed together; the output is then gated back to the TP control. At the same time, each gate to the dot-or is also enabled by the select signal from the Name Bus Compare Logic. To check an NFR, then, it must first be selected and the NFCK/E signal activated to gate the selected NFR to the dot-or. The return signal to control is called NFOE and = 0 if the interrogated NFR = 1.

The two 4-bit counters, CCT and ACT are used to keep track of the names which have been assigned in the name permutation process. As will be explained in section 5.5.3.2, CCT is used to store the next name to which a PR will be renamed. It is set to zero and then incremented each time a PR is renamed. ACT is used after the tag sequence has been exhausted to check the old PR names. The logical diagrams of the two counters can be found in drawing number 15-1 of the Taxicrinic Processor Logic Book.

The CEQA signal from these counters to the main control is used in the permutation sequence to indicate that the CCT = ACT.

5.5.3.2 Name Permutation Sequence Description

The name permutation control registers are set up as shown in Figure 5.5.3.2/1. Each physical PR has associated with it a NR, SNR and NFR. The various gate paths between each group of 3 registers and the Shadow Name Bus are enabled by the fifteen select lines, NCE_i (i = 0, 1, . . . , 14) coming from the Name Bus Compare Logic. When an NCE line is selected, it enables all of the gate paths involved in its corresponding NR-SNR-NFR group. Accordingly, when a gating signal is given to all 15 groups only the group whose NCE line is "on" actually does any gating.

The detailed renaming sequence proceeds as follows. Initially the CCT counter is used to keep a running count of the number of the phrase which is presently being processed (0, 1, 2, . . . 13). The TGR is placed on the Name Bus and compared with each Name Register simultaneously to give an output on the NCE select line corresponding to the PR-NR-SNR-NFR combination which has this name. The control then gates the CCT to the Shadow Name Bus (SNB) and gates the SNB to the selected SNR but only if the NFR (TGR) has not been previously set to one. In ~~this~~ case, the NFR (TGR) is set to 1 while the SNR is being loaded. Then CCT is incremented by one and the next phrase, if any, is processed. However, if the NFR (TGR) is already one, the current name in the TGR has already been used in the renaming process and, therefore, must be ignored as far as the renaming is concerned. In this case, the SNB is not activated and the SNR is left unchanged, the CCT is not incremented and the next phrase, if any, is processed.

Since only the gates belonging to the selected group have been enabled, its SNR and NFR are the only registers affected. Thus prior to processing of the next phrase, the group indicated by the value of the tag of the last phrase processed has its old PR name in its NR, its new PR name in its SNR, and its NFR set to one to indicate that a new name has been assigned. This process is then repeated until all the tags in the imprimitive instruction have been scanned.

Figure 5.5.3.2/1 Name-Permutation Control Registers

The second part of the name permutation sequence now begins with the ACT counter set to zero, while the CCT counter retains the first name not used as a "new" PR name. The ACT counter is used to check the lower numbered PR names to be sure they have all been changed during the permutation process. After each name has been checked, ACT is incremented and is compared with the CCT. If they are equal, the process is finished.

In this checking procedure ACT is gated onto the Name Bus to be compared with the NR's. This activates the NCE select line belonging to the PR whose old name is equal to the value of the ACT counter. The NFR of this PR is then checked by activating NFCK/E and observing the output from the dot-or connection mentioned in Section 5.5.3.1. If the selected NFR is 1, the dot-or output line to the control will be zero. In this case, the ACT counter is incremented by one and if ACT and CCT are not then equal, the process is repeated with the new value.

If the selected NFR is 0, i.e., the PR was not renamed during the first part of the permutation sequence, the CCT is used to find the next name in the tag sequence which is available. Since the new name must be one which was in the tag sequence, the name in the CCT is gated to the NB and checked to see if its NFR is 1. If it is, it was renamed by the tag sequence and can therefore be used as a new name for the ACT. (In the example permutation of Section 5.5.3, just this situation prevails: old PR #1 is renamed PR #5). If it is not, then that particular PR was not renamed by the tag sequence and, therefore, it must retain its previous name.

At this point, the control sequence departs from the description given in Section 5.5.3. In order to keep from using a 3rd counter, the CCT is used to search the NR's greater than $i-1$, where i is the number of PR names in the tag sequence. However, when CCT is incremented during this search, the count of the number of names in the tag sequence is lost. The end of the sequence can still be determined when $ACT = CCT$, if the SNR of the PR's named by the CCT

as it is incremented are loaded with the old name if they have not already been changed. In this manner each NFR of the PR's whose names are between $i-1$ and the value in CCT will be one, either because the names were changed during the processing of the tag sequence or because they were changed while the CCT was "looking" for names "greater than" $i-1$ which were in the tag sequence. Thus when the ACT has finished permuting all of the pointer names between 0 and $i-1$ it will proceed through the remaining pointers until $ACT = CCT$ without any further renaming because all of the intervening NFR's already equal 1.

In the remaining process the highest count attained by CCT and ACT is equal to the "largest" name input in the tag sequence. Thus, in the example of Section 5.5.3, where the largest name input was 11, $CCT = ACT = 11$ would be the last cycle completed before the SNR's were conditionally gated into the NR's.

When ACT equals CCT, the final step is ready to be performed. All of the SNR's are gated into their corresponding NR's conditional to the corresponding NFR being on. If the NFR of some NR is "0", the contents of the NR remain unchanged (the NR has not been renamed). Thus the "new" PR names now become the "old" PR names.

The detailed flow chart for this sequence is given at the end of this section.

Note that the reverse process, which must be performed after the imprimitive instruction has been completed, is exactly the same except for two places where the roles of the CCT and ACT are interchanged. Note that in this name repermuation process the name flag registers are again set according to the tag fields in the operand phrases. This again enables us to be able to check for multiple appearances of the same tag. The CCT is used to count the phrases and indicates which PR is currently being renamed.

At the end of the postscan the name repermuation is completed by again using the ACT counter to see which names have been assigned as new names. Every time a name is found which has not been used as a "new" name, it is used as the new name for the PR currently indicated by the CCT, the NFR (ACT) is set to 1, and the CCT counter is incremented. Eventually, ACT will become equal to the CCT and at this point the process terminates. Note that exactly the same type of operations were performed in the prescan; accordingly, the resulting new names are identical to those before the call or execute instruction.

Name Permute/Repermute Sequence Flow Chart

5.5.3.3 Name Permutation Control Logic

The Name Permutation Control Logic is really broken down into two sections. The start of the sequence is so intertwined with the main imprimitive sequence that it is given in the section on imprimitive control logic (Section 5.5.4.3.) with no attempt made to separate the two.

The end of the Name Permutation sequence is fairly independent of the Imprimitive sequence so that it can be separated. Also it turns out to be rather difficult to make an efficient control design using only control points so a special hybrid design was used which utilizes a few control points along with several flip flops and time delay circuits. A control point flow chart of sorts is given at the end of this section while the logic itself is shown in Figure 5.5.3.3/1.

The main timing problem occurs in the test of the Name Flag Register. In order to test an NFR it is necessary to load the Name Bus with a number and turn on the NFCK/E signal. After the selection logic has settled down the NFOE signal will be on if the specified name flag register was 1. However if we were to try to do this with a conventional control point we would run into one of two problems:

- 1) If the NFOE signal is checked after a control point which loads the Name Bus and turns on the enabling signal, these signals will be turned off and the NFOE signal will no longer be valid for the check

- 2) If the NFOE signal is checked during the control point operation, it will not be valid until a certain time after the task has begun and the selection logic has settled down. Thus the initial value of NFOE will not be valid and may give incorrect results.

The obvious solution to this problem is to delay the activation of the check for NFOE until the selection has stabilized. This could be done by using a control point as a delay and operating it in parallel with another one which activates the selection logic and keeps it turned on until after the choice has been made. However a way which uses less logic is that shown in Figure 5.5.3.3/1.

Figure 5.5.3.3/1 Name Permutation End Sequence Control Logic

In Figure 5.5.3.3, IMT66 and IMT67 are task signals which are controlled by flip flops instead of standard control points. After a suitable delay time (Δ), the selection logic has settled down so that a decision can be made as to which of the following flip flops is to be set. When either one has been set the flip flop controlling IMT66 (or IMT67) can be turned off.

The other main problem is to be sure that all the flip flops and control points are properly set when the sequence is not in use. The control points can be controlled in the usual way by connecting their set signals to the initiating signal. The flip flops are controlled by letting $\overline{\text{SET}}$ be one of the signals which resets each flip flop.

Name Permutation End Sequence Control Step Flow Chart

10/28/68

Section 5.5.3.3 - 1/1

5.5.4 Imprimitive Sequence

5.5.4.1 Imprimitive Sequence Description

The purpose of this section is to combine the details explained in the previous sections into one coherent whole. The result is the set of flowcharts at the end of this section. For the purposes of this section, all of the interrupt information will be ignored. This will be explained in Section 5.5.4.2.

In the overall view, the flowchart can be broken down into three main operations:

- 1) Initialize the stacks and SBR on the basis of the type of instruction and then perform the necessary operator phrase pre-operations.
- 2) Scan the operand phrases and perform the necessary pointer operations and name changes.
- 3) After the scan is finished, complete the name permutation if necessary, complete the stack arrangement, and possibly set up for a rescan if a conditional subtraction failure has taken place.

These operations are shown on pages 1, 2 and 3 respectively of the flowchart.

The sequence begins by setting the scan control flip-flop, RSCN, the name permutation flip-flop, PER, and the execute flip-flop, EXEC, according to the type of instruction which is being performed. If the instruction is a CALL, EXECUTE, or GOTO, the SBR is loaded with the contents of PR#0 at this time so that the initial address of the instruction will be saved.

If the instruction is an EXIT, PR#0 is popped once and the new address is used to reload the IBR and IR. If the instruction at this address is a CALL, then control is transferred to entry point A of the IMPRIMITIVE sequence. Otherwise PR#0 is saved in the SBR while the PR#0 stack is popped again. The SBR is then loaded into PR#0 before transferring control to the Final Control sequence.

At entry point A, the ICT is checked to see if the IR only contained the mnemonic byte and, if so the IBR is reloaded. Then ICT is incremented and the IR is loaded with the zeroth operand phrase.

At this point the PR#0 modification detector flip-flop, PROMD, is reset along with the CCT, the ACT and all of the name flag registers. The tag register is loaded with the tag field from the IR. If the phrase is long and the IBR does not contain all of it, then the IBR must be reloaded and the needed bytes merged into the IR. If the operation under way is part of a rescan, the rest of the initial operations can then be skipped.

If the operation is not part of a rescan of an instruction and PER = 1, then we must be in the pre-scan of a CALL or EXECUTE and it will be necessary to set up the new PR#0 stack with the proper return parameters as explained in Section . To do this the stack indicated by the tag register is first pushed once. If the tag happens to be 0, this means that the address of the current instruction +1 will be pushed into the stack instead of just the address itself (remember the ICT has been incremented by 1). This will present no problems later on however since on a rescan of a call with an initial tag field of 0, the value stored in the stack at this point is never actually used for anything except a place holder.

The next step in a CALL or EXECUTE instruction is to exchange the SBR with the PR indicated by the tag and then pushing the stack once again. This causes the return address of the instruction to be saved in the stack (remember the SBR contained the initial address of the instruction).

Finally the values and flags of the SBR and the PR mentioned by the tag are exchanged again. At this time, the PR will be set according to whether or not the instruction is an EXECUTE by using the SETEX option in the Exchange PR-SBR sequence. Then the pre-operations are performed for the first operand phrase.

If the current operation is not a rescan and if the instruction is a GOTO instead of a CALL or EXECUTE, there is no need to fool around with the stacks and therefore the pre-operations are performed right away. Note that the SBR will be saving the initial byte address of the instruction for the remainder of the execution of the sequence. If any phrase (including the zeroth) has a tag of zero, the SBR and PR#0 must be exchanged while the Phrase Sequence is being executed since all modifications of PR#0 are assumed to be performed with reference to the initial byte address of the instruction. However, at all other times in the imprimitive sequence PR#0 is used to direct the scan and, therefore, will not contain the initial address and must not be modified except in the scanning process itself. By setting CKTGZ = 1 a PR-SBR exchange will be made if the TAG = 0 and any modifications are made to the PR by the phrase being processed. At the end of the modifications they will be exchanged back.

The final operations on the first operand phrase consist of loading the various shadow name registers and setting the proper name flag registers. If the instruction is a CALL or EXECUTE during a pre-scan the SNR and NFR indicated by the TGR are set to 0 and 1 respectively. In the prescan of a GOTO, the SNR's indicated by the TGR and CCT are set to the values held in the CCT and TGR respectively while both NFR's are set to 1 (i.e. the names of the PR's indicated by the TGR and CCT are exchanged). In the rescan of an instruction, the name in the TGR is loaded into the SNR of the PR indicated by the CCT.

Finally, the CCT is incremented to 1 and control passes to entry point B. At this point, the ICT is incremented according to the number of bytes in the phrase by calling INCR ICT. Next, a check is made of the continue bit. If it is one the operand phrase processing sequence is repeated. If it is zero the cleaning up procedures are begun. These consist of looking for an interrupt and checking the ICT for 0 and reloading the IBR if necessary.

The subsequent operand phrase processing sequence employs part of the first operator processing sequence for its final operations. However its initial stages are different and much simpler. The Phrase Process sequence is used to process the phrase but SICT is set to 1 so that the actual phrase sequence is not performed. Next the tag field of the phrase is checked if the instruction is a CALL or EXECUTE. In these instructions if the PR named by the tag field of the operand phrase has been mentioned before, all processing of the phrase is skipped and the next phrase is processed.

Next a check is made to see if we are in a rescan and that the "popping" bit of the phrase is off. In this case there is no operation to be performed. Otherwise the Phrase sequence must be executed if we are in a pre-scan and the unstack PR sequence if we are in the post-scan. Note that in the rescan the CCT is used to indicate the stack being popped since at this point the pointer registers have the new names that were given to them at the end of the pre-scan. It should also be pointed out that since no pop is allowed on the first phrase of the instruction and since this part of the sequence only deals with phrases other than the first, there will never be any need to exchange the SBR and PR#0 during a rescan, as may be the case in the pre-scan.

After the modifications have been completed, the permutation flip-flop must be checked. If the instruction is an EXIT or JUMP there is no permutation of names and therefore the actions concerning the Shadow Name Registers and the Name Flag Registers are skipped. If PER = 1, and we are in the pre-scan, the Shadow Name Register indicated by the tag will be loaded with the contents of CCT. If we are in a post-scan the SNR indicated by the CCT will be loaded with the tag. The CCT is incremented each time a phrase is processed. The Name Flag Register indicated by the tag is set to one after which the ICT is incremented and checked for zero. Then we are back to checking the continue bit.

If the phrase just processed was the last one in the instruction then control passes to entry point C. If we are rescanning or if the instruction is a CALL or EXECUTE, the name permutation must be completed. If the instruction is an EXIT or JUMP this can be skipped since there has been no name permutation. Since the completion of the name permutation has been described in detail in Section 5.5.3.2 it will not be described here.

Completing the stack arrangement can be divided up into three subsections.

- 1) The rescan completion sequence
- 2) The operand failure sequence
- 3) The no operand failure - CALL, EXECUTE and GOTO sequence.

If we are ending a rescan then the current value of PR#0 is saved in the SBR to be used as the address of the next instruction. Then PR#0 is unstacked, the SBR and PR#0 are exchanged and control is given to the final control sequence.

If there was an operand failure then there are two cases: if the instruction is a GOTO there will be no rescan and therefore all that is done is to set the conditional subtransaction failure flip-flop before returning to the final control sequence. If the instruction is a CALL or EXECUTE then in addition to setting the CSBF flip flop, the names will be permuted, PR#0 will be unstacked and the IBR and IR will be reloaded from the address thus exposed. Finally the RSCN flip flop will be set before returning to entry point A of the imprimitive sequence.

If there is no operand failure, the SBR is loaded into PR#0, the names are permuted, PROMOD is reset to 0 and the IBR is reloaded. If the instruction is a GOTO control is given to the final control

sequence at this point. If it is a CALL or EXECUTE, a check is made to see if the OS must be reinitialized and a check is made for interrupts before returning to the Main Control Sequence. The reason control is not given to the Final Control Sequence is that the Final Control Sequence always checks the EXECUTE flip flop and since this must not be done after performing a CALL or EXECUTE, a separate sequence is used.

Imprimitive Sequence Flow Chart

Imprimitive Sequence Flow Chart

10/28/68

Section 5.5.4.1 - 2/4

Imprimitive Sequence Flow Chart

Imprimitive Sequence Flow Chart

5.5.4.2 Imprimitive Sequence Interrupt Description

The Imprimitive sequence interrupts are complicated by the fact that the sequence cannot always be restarted after an interrupt has been discovered. There are actually 6 possible interrupt return points.

In the initial part of the EXIT instruction sequence an interrupt may occur in the UNSTACK PR or IBR Reload sequences. If an interrupt occurs in the UNSTACK PR sequence the whole instruction may be restarted. If it occurs in the IBR Reload sequence PR#0 is restored to its old value by loading it from the SBR before the interrupt sequence is started. When the interrupt returns the whole instruction may be restarted.

If the location to which the EXIT returns does not contain a CALL instruction, an illegal EXIT interrupt is caused. PR#0 will be loaded with the address of the original EXIT instruction before the interrupt sequence is started.

In the main part of the sequence which processes the first phrase if an interrupt occurs in the first or second IBR Reload, the interrupt must return to interrupt return point L upon completion. The instruction cannot be restarted since it may have been an EXIT. Note that in the second IBR Reload, if there is an interrupt the ICT must be returned to its original state by adding 7 (i.e. subtracting 1 if the overflow is ignored).

If an interrupt occurs in the first Stack PR sequence the whole instruction can be restarted since the control could only have gotten to this point if the instruction were a CALL or EXECUTE. In this case the ICT must be restored to its original value before the interrupt sequence is entered.

If an interrupt occurs in the second STACK PR sequence call, the interrupt must return to interrupt return point M.

In the main Imprimitive loop on page 2 of the flow charts, interrupts may occur in the Phrase Process or Unstack PR sequences. Interrupts which occur in the Phrase Sequence will be handled by it and will not be "bucked up" to the Imprimitive sequences. All of the Phrase Process and Unstack PR sequences cause returns to interrupt return point N.

In the final part of the Imprimitive Sequence on page 3 of the flowcharts all interruptible sequences will return to interrupt return points P, Q, or R. Interrupt return point Q and R are needed because it is impossible to undo the name permutation.

5.5.4.3 Imprimitive Control Logic

The Imprimitive Sequence Control Logic is fairly complicated, if only because there is so much of it. It is initiated directly by the Main Control Sequence whenever an imprimitive mnemonic is detected.

There are a variety of control flip-flops used in the sequence. They are listed below. Note that since the Imprimitive sequence may be interrupted during its execution, the state of these flip-flops must be saved as part of the status of the machine.

EXEC: indicates an EXECUTE instruction is being executed.

RSCN: equals 1 when a rescan (or post-scan) of an imprimitive instruction must be performed.

PER: equals 1 for the CALL and EXECUTE instructions and indicates that the names must be permuted.

The initial exit entry shares all of its control points with parts of the sequence at the end. Note that control points IMT32 and IMT33 are actually used in 3 places the third place being the initial sequence of the CALL, GOTO, and EXECUTE instructions.

In control point IMT38 it is not certain at the present time whether it will be possible to start the unstacking sequence while the TGR is being loaded. This is due to the fact that the UNSTACK PR sequence's first control step involves the contents of the TGR. If a detailed look at the logic shows that there will be a safe delay between the beginning of the IMT38 task signal and the beginning of the first control step of UNSTACK PR so that the TGR will contain valid information at that time then it will be alright to leave the two operations in one control point. If this is the case then it would also be possible to combine IMT33 and IMT34.

The same sort of situation occurs in IMT42 and IMT43 where it is not certain how long a delay will occur before the IBR RELOAD sequence needs the names which are being permuted by IMT42. If there is a long enough inherent delay in the control point calling mechanism, these two control points could be combined.

The imprimitive sequence contains a great many decisions and as a result the decision logic is fairly complicated. I have tried to follow the general rule of doing all decisions in parallel if possible. However in this sequence to follow such a course would entail a great deal more logic. Therefore in many places speed has been traded for logic economy by dividing the logic into stages and generating gating signals which are then used to drive another stage of decision logic.

The decision logic after IMT5 is an example of this. The equations for this logic are given in Figure 5.5.4.3/1. Note that the logic forms the signal IMG1 which is then used to drive more decision logic.

The equations for the decision logic after IMT18 and IMT19 are shown in Figure 5.5.4.3/2.

The equations for the decision logic after IMT21 are shown in Figure 5.5.4.3/3. Several short cuts should be mentioned about these equations. Since many of the tests occur in several places it sometimes happens that certain paths are impossible as long as the control flip-flops do not change state during the sequence (which should never happen). As a result, for example IMT27 cannot "drive" IMT17 since in order to get to IMT27, RSCN must be off and to get from IMT27 to IMT17, RSCN must be on.

The same type of thing is true for trying to drive IMT17 from IMT29 and IMT30 from IMT28.

$IMA05 \cdot (ICTEQ6 \vee ICTEQ7) \cdot IR9$

IMT6

$IMA05 \cdot (\overline{IR9} \vee \overline{ICTEQ6} \cdot \overline{ICTEQ7})$
 $\vee IMA06$

IMG1

$IMG1 \cdot RSCN$
 $\vee IMA016$

IMT17

$IMG1 \cdot PER \cdot \overline{RSCN}$

IMT8

$IMG1 \cdot \overline{PER} \cdot \overline{RSCN}$
 $\vee IMA011$

IMT13

Figure 5.5.4.3/1

Decision Logic Equations After IMT5

(IMG3 v IMA027 v IMA028) • $\overline{\text{PER}}$
v IMA024 • NFOHE

IMT20

IMA020 • $\overline{\text{IR8}}$

(see Figure 5.5.4.3/4)

Go to C

IMA020 • IR8 • inter

Go to Interrupt

IMA020 • IR8 • inter
v interrupt return

IMT21

Figure 5.5.4.3/2

Equations for Decision Logic after IMT18 and IMT19

IMA024 • $\overline{\text{NFO}}\overline{\text{NE}}$ • $\overline{\text{RSCN}}$
v IMA021 • $\overline{\text{PER}}$ • $\overline{\text{RSCN}}$

IMT27

IMA024 • $\overline{\text{NFO}}\overline{\text{NE}}$ • $\overline{\text{RSCN}}$ • $\overline{\text{IR6}}$
v IMA021 • $\overline{\text{PER}}$ • $\overline{\text{RSCN}}$ • $\overline{\text{IR6}}$

IMT28

IMA021 • $\overline{\text{PER}}$ • $\overline{\text{RSCN}}$ • $\overline{\text{IR6}}$
v IMA024 • $\overline{\text{NFO}}\overline{\text{NE}}$ • $\overline{\text{RSCN}}$ • $\overline{\text{IR6}}$

IMG3

IMA027 • $\overline{\text{PER}}$

IMT30

IMA028 • $\overline{\text{PER}}$
v IMG3 • $\overline{\text{PER}}$

IMT17

Figure 5.5.4.3/3

Equations for Decision Logic After IMT21

It should be noted that in IMT24, the TNB/G and NFCK/E signals must remain on while the test of NFOE is being performed. This is the same problem as noted previously in the name permutation end sequence (see Section 5.5.3.3) and is handled in the same way.

Control points IMT26 and IMT29 could be combined if automatic resetting control points are used. There would be a consequent increase in the decision logic, however, and this may turn out to cancel the savings in logic.

The C entry point in the imprimitive control logic is represented by the expression $IMG2 \cdot \overline{IR8}$. Figure 5.5.4.3/4 gives the equations for the decision logic after entry point C. Since this is really an extension of the equations in Figure 5.5.4.3/2, the $IMG2 \cdot \overline{IR8}$ expression is used instead of forming a new signal and wasting another collector delay.

IMG2 • $\overline{IR8}$ • (RSCN v PER)	IMT31
IMA031 • RSCN	IMT32
v IMA040 • $\overline{\text{call inst. (see text)}}$	
v IMA01	
IMG2 • $\overline{IR8}$ • $\overline{(RSCN v PER)}$ • OPANDF	IMT37
v IMA031 • RSCN • OPANDF	
IMA037 • PER	IMT38
EXIT ENTRY	
IMG2 • $\overline{IR8}$ • $\overline{(RSCN v PER)}$ • $\overline{OPANDF}$	IMT41
IMA031 • RSCN • OPANDF	

Figure 5.5.4.3/4

Equations for Decision Logic After Entry Point C

Imprimitive Sequence Control Step Flow Chart

Imprimitive Sequence Control Step Flow Chart

Imprimitive Sequence Control Step Flow Chart

Imprimitive Sequence Control Step Flow Chart.

5.6 System Instructions

The Supervisory Instructions are those instructions which are related to operations involving some contact with the Operating System. Some of these instructions are protected instructions and may only be used by the Supervisor task. Others are to be used by the user tasks to obtain Supervisor assistance or to use features of the Operating System.

5.6.1 Supervisor Call

The Supervisor Call Instruction (SVC) transfers control from the currently running task to the Supervisor task. Its main purpose is to restrict access to the Supervisor in such a way that the Supervisor can protect itself from unauthorized use. In addition the SVC instruction must save all of the Pointer Registers of the calling task in the Status Block provided for that task in the Supervisor's Task Status Segment.

In performing an SVC instruction the first operation is to obtain the descriptor for the Supervisor task's Segment Table by making a 1 word read access to the System Constants Segment. This segment is given the internal segment name 8 in every task in the system. The problem which makes this access difficult is that it must be made without permanently modifying any PR's and without temporarily storing anything in core memory. Thus the first step is to temporarily save the Segment Name field of PR#0 in the IR so that Segment Name Register #0 can be used to store the Segment Names which will have to be used in the coming operations. Then the number 8 is loaded in Seg. Name Register #0 and the Segment Name Select Register is set to select #0.

Next the DR is loaded with the address of the Supervisor's Segment Table descriptor within the System Constants Segment (in this case 0) and the Memory Modified Entry Sequence is activated. When the sequence returns, the descriptor contents will be in the LR. This is then gated to BR#0 and all of the OK flip-flops on the Associative Registers are set to zero.

The next operation to be performed is to save the calling task's Pointer Registers. Once again Segment Name Register #0 must be used to temporarily store Segment Names. First it is loaded with the internal segment name of the Task Constants Segment for the Supervisor (in this case 16). Then the DR is loaded with the address within the Supervisor's Task Constants Segment of the internal segment name of the Task Status Segment. After a Memory Access has been performed, the Memory Read Sequence will return with the Task Status Segment's internal segment name in the LR.

Next the LR contents are loaded into Segment Name Register #0. Then the contents of the Task Register are shifted left an appropriate number of bits to allow for the size of a Status Block and the result is the virtual address of the particular calling task's status block. This address is loaded into the DR to be used as the address for storing the contents of PR#0. A write modified access to the memory sequence is performed (still using SN Register #0 as the selected SN) and when the addressing sequence returns PR#0 is loaded into the LR and the IR (which has been saving the original contents of SN Register #0) is loaded into the DR. Then the Memory Write Sequence is called.

Once this sequence returns, it is only necessary to increase the AR by 8 and to load the next PR and S. N. Register into the LR and DR and repeat the sequence. This continues until all of the Pointer Registers have been stored.

Finally the TP again accesses the System Constants table (after loading SN Register #0 with 8 and the DR with 4) to obtain the PR#0 value and segment name field which must be used to give control to the Supervisor SVC validity checker. Once these values have been loaded, all of the PR link fields are set to zero and the Protected Instruction flip-flop is set. Then the instruction operations are finished.

5.6.2 Supervisor Return

This protected instruction is used by the Supervisor to return control to a user task. The user task being activated is determined by the current contents of the Task Register.

The first step consists of accessing the Task Status Segment to reload the Pointer Registers. In order to do this the contents of the Task Register must be obtained and shifted an appropriate number of bits to allow for the size of a Status Block. Then this address can be used to obtain the PR contents for the user task being given control. Except for the case of PR#0 all accesses are conventional read-load register types. For PR#0 the PR contents are loaded as usual, but the contents of Segment Name Register #0 are temporarily stored in the IR since that register is still needed for other accesses.

Once the PR's have been loaded, the next job is to load BR#0 with the new tasks Segment Table descriptor. This is done by loading the contents of the Task Register into Segment Name Register #0 and the DR with zero and then calling the Modified Entry Memory Sequence with LOC=1. This will cause the base descriptor of the new task to be loaded into a base register if it was not already in one. Once this has been done the Associative Logic can be used to read out this base information and then store it in BR#0.

Finally, the SN field contents of PR#0, which has been saved in the IR, is loaded into Segment Name Register #0, the OK bits on the Associative Registers are all set to zero, and the Protected Instruction flip-flop is reset to zero. Then control is sent to the Final Control Sequence.

5.6.3 Rename

The Rename Instruction, RNAM, is a protected instruction used by the Supervisor to assign a given segment a known name in the Supervisor Segment Table's Dynamic Segment region. The instruction utilizes two operand phrases and the Operand Stack. The first operand points to a base descriptor while the second operand indicates an available space file.

The first step in the instruction is to prepare the Operand Stack so that the new name can be pushed into it. This entails a call to OS ENTRY. Next a call to AS GET is made to obtain a double word cell in which to store the base descriptor. The address of this cell is pushed into the OS. Then the base descriptor is read in from core and stored into the previously obtained cell. Finally control is sent to the Final Control Sequence.

5.6.4 . Sleep

The SLEEP instruction is a one operand protected instruction used by the Supervisor to idle the TP. The operand points to the address of a halfword cell containing the number of tenths of milliseconds the TP will remain idle before reawakening. During the time in which the TP is in the idle state it is possible to awaken the TP only with an external activate interrupt.

The hardware implementation is fairly straightforward since all that is necessary is to place the control logic in a state where it will wait for either an external activate interrupt or for the time to expire. If the time expires, the TP will begin execution of the final control sequence and then go to the next instruction. If it is interrupted, it will go to the interrupt sequence. It should be noted that since the sleep timer is not part of the information retained during an interrupt, when an interrupt return is finally performed, the sleep timer will be set to zero and thus the TP will immediately wake up.

5.6.5. Activate TP

This protected instruction is used by the Supervisor task to cause a given TP to begin processing a new task. In actual fact it is four instructions since the operation code contains a two bit field which designates which TP is to be activated.

The Activate TP instruction causes the TP executing the instruction to make a TP Interrupt request to the Interrupt Unit. This request specifies which TP is to be interrupted. Eventually the Interrupt Unit will notify the requesting unit that the TP being activated has received the interrupt command. At this point the TP executing the Activate TP instruction enters the Final Control Sequence and then proceeds to the next instruction.

As far as the TP receiving the activate command from the Interrupt Unit is concerned, the command is treated as a normal external interrupt. The Interrupt Handler is responsible for detecting the type of interrupt and transferring control to the Supervisor Reactivation Procedure.

5.6.6 Increment and Check

The Increment and Check instruction, INCK, is used to update control words of a specified format. It is a one operand instruction with the operand indicating the control word to be updated.

The control word itself is divided into four fields as shown below:

□	□	□	□	□	□	□	□
Initial Value	Increment Value	Maximum Value	Pointer Value				

The INCK instruction first accesses the control word in memory via the Exchange Net. Then, while still maintaining the data path to the core memory, it increments the control word pointer value field by the value contained in the increment value field. This new value is then compared with the maximum value field in the control word. If they are not equal, the EQ indicator is set to 0 and the newly calculated value of the control word pointer is written back into the control word along with the unchanged maximum value field. Then the Exchange Net is released.

If the new control word pointer equals the maximum value, the same procedure is followed except that zero is stored in the control word pointer and the EQ indicator is set to 1.

This instruction has two important uses. In the handling of interrupts it is used for checking the interrupt buffer for interrupt data to be processed. In this case the control word to be used is the Interrupt Storage Segment's Last Emptied Block control word which is located in the status block of the Interrupt Storage Segment (see Section 5.6.4.2). In this case the pointer is originally set to zero, the maximum value field is set to the initial byte address within the Interrupt Storage Segment of the last usable interrupt storage block, and the incrementation value is the length of one block of storage.

By using this instruction the system ensures that two TP's (running in Supervisor Mode) will not try to process the same interrupt information.

The second use of this instruction is for checking the control word of so-called "critical sections". In this case the control word pointer only has two values, 0 and 1, while the maximum value field contains the number 1. The 1 pointer value corresponds to the case where no processor is in the critical section and the 0 value corresponds to the case where a processor is in the critical section. The incrementation value is zero.

When a processor wants to use a critical section, it first performs a INCK on the control word for that section. If there is no processor in the section, the pointer value will be 1 and after being incremented by 0 it will equal the maximum value field, thus causing a 0 to be returned to the field. Then after testing EQ and discovering it is on, the TP can begin processing in the critical section.

Now suppose a second TP wants to process in this same area before the first TP is finished. In this case it will also perform an INCK. This time however, the new pointer value, after being incremented by 0, will still be zero and will thus not be equal to the maximum value field of 1. When this second TP checks the EQ indicator it will be off and thus the second TP will have to "go to sleep" for awhile.

Eventually the first TP will finish its processing and at this point it will set the control word pointer value back to 1. Thus eventually when the second TP again checks the control word, it will be allowed to use the "critical section".

5.6.7 OK OFF

This instruction is very straightforward. It merely resets the OK bits on the seven Associative Registers connected with the TP Base Registers.

5.6.8 Link

The LINK instruction is a two operand instruction designed to facilitate dynamic segment linking in the Illiac III system. The first operand is generally short and refers to the PR which is pointing to the desired linkage table (see Section 5 of the Programming Manual). The second operand is generally long and refers to a Pointer Register which contains the relative address of the entry in the linkage table for the external segment address to be linked. As discussed in Section 4.2 of the Programming Manual this relative address is assigned at the time the segment was compiled or assembled and is usually loaded into the PR by means of a replacement operation on the second operand phrase of the LINK instruction. However, there is no hardware restriction on the length of the LINK operand phrases other than the fact that at least one of them must be short.

The instruction begins by adding the PR value field associated with the second operand phrase to the PR value field associated with the first operand and using the resulting sum along with the segment name associated with the first operand, to access the 1 word linkage table entry. If the 1st flag on this word is a '0', a linkage interrupt is performed and control passes to the Interrupt Sequence. If the flag is '1', the entry has been filled and the word of data is loaded into the PR specified by the second phrase and the instruction is finished. At this point the PR specified by the second operand phrase contains the internal segment name and virtual address within that segment of the entry point originally specified by the external reference. From this point on this PR can be used to reference any data in that particular segment.

5.6.9 Who

The Who instruction provides a means for a program to find out which TP it is running on. It has one operand phrase which indicates which pointer register is to be loaded with the number of the physical TP which executes the instruction. This instruction is useful for a variety of Supervisor functions which require knowing information about particular processors (e.g. priority storage and retrieval).

The implementation is very straightforward. The status of the control switches is loaded on to the PRSNB bus and from there gated to the Pointer Register indicated by the operand phrase.

5.6.10 Set Interrupt Mask

This instruction is used to set particular states into the Interrupt Mask Register. It presents a few problems since in order to be at all efficient it must not be a protected instruction. However, if it is not a protected instruction a task could conceivably use it to inhibit all interrupts once it got on the machine and thus keep from being ever thrown off due to an external activate or timer interrupt.

The final solution was to allow the Set Interrupt Mask instruction to have control over all interrupts except the following: timer interrupt, external activate interrupt, and external I/O. Only the hardware can mask out these interrupts when control is first given to the Interrupt Handler. Whenever a SIM instruction is executed these interrupts are automatically unmasked. Thus it is possible for the Interrupt Handler to have all interrupts masked out during the initial period of interrupt processing when this is necessary. Since the Interrupt Handler is in read/only storage it will not be possible for the user to "take over control" before the critical interrupts have been unmasked.

A one in a given position of the PR value field sets the corresponding mask to one, which inhibits the immediate recognition of that interrupt.

5.6.11 Reset Interrupt Mask

The Interrupt Return instruction, INRT, has no operands but uses PR#1 as an implied operand pointing to the Interrupt Storage Block containing the return status information. The execution of this instruction will cause the TP hardware to restore its status before the interrupt. Just prior to resuming operation, the TP will also restore the Interrupt Storage Block to the available space list.

5.6.12 Reserve Unit

The Reserve Unit Instruction, RESU, is a protected two operand instruction used by the Supervisor to reserve a particular unit, attached to one of 18 local exchanges on the Exchange Net, to one of 6 processors. The first operand indicates a Pointer Register containing a number from 0 to 5 and indicates the processor sort involved in the reservation. The second operand indicates a Pointer Register containing a number from 1 to 18 and indicates the local exchange involved in the reservation.

This instruction is necessary for those cases in which a particular processor desires to use a particular unit over a long period of time without interference from other processors. This may be necessary because of minimum response time requirements (e.g. in the case of an AU) or because intermediate data pertinent only to a particular task will be contained in the unit over a long period of time (e.g. in the case of a PAU).

The hardware implementation of this instruction is not clear at this time. It will certainly involve a certain amount of work on the part of the Exchange Net.

5.6.13 Load Task Register

This protected instruction loads the value field of the PR indicated by the operand phrase into the Task Register.

7/17/70

Section 5.6.13 - 1/1

5.6.14 Store Task Register

The Store Task Register instruction, STR, is used to save the name contained in the Task Register in core. It loads the Task Register contents into the LR and then writes the LR into the halfword memory location indicated by the operand phrase. There is no immediate option for this instruction.

5.6.15 Interrupt Return

The Interrupt Return instruction is very straightforward given the existence of the Interrupt Return sequence explained in Section 4.6.1.5 of Volume II of this manual. It's single operand indicates the location of the interrupt block to be used in restoring the previous TP status.

5.6 - Appendix Miscellaneous Supervisor Problems

The purpose of this section is mainly historical. In the process of working out the many Supervisor problems which have come up, many people have developed a variety of solutions and partial solutions which, for one reason or another (or sometimes for no reason) were finally discarded. However since a great deal of thought went into these problems and since most of the people who have done the work have either forgotten it or moved on, it seems desirable to record these ideas so that if conditions change they will not have to be reinvented again. Thus the purpose of each of the following sections is to illuminate a particular Supervisor problem, discuss the various possible solutions, and then explain why the eventually chosen solution was picked.

5.6.A.1 The Lockout Problem

In any multi-processing system one of the objectives of the system is to allow common sections of code to be shared. For instance it should be possible for several processors to execute the Assembler at the same time. (In many respects this is trivial to accomplish in Illiac III, since each processor has its own set of registers. However, it must be observed that the memory arrangement is not well suited to concurrent access of routines because it is not interleaved.)

In spite of this, certain sections of the Supervisor must be processed in a serial fashion. For instance, if a section of the Supervisor requires that a table be altered, this section of code cannot be processed in parallel. Thus it must be possible for a TP entering such a section of code to 'lock-out' any other TP's which may subsequently attempt to execute this section, until such time as the first TP has finished (and has 'unlocked' the section).

A variety of solutions to this problem were suggested. All of them involved using control word to indicate whether or not a certain critical section could be used. The trick then became to check and modify the control word in one operation so that if one TP began accessing the control word, no other TP could get to it until after the 1st TP had looked at it and then modified it if it did not already indicate that some other TP was using the section. Since this operation could easily be accomplished simply by refusing to drop the Exchange Net request line once access had been granted to a core box, the differences between the various solutions centered around the format of the control word.

One important point concerning all of the methods to be discussed should be emphasized. Unfortunately these procedures are not guaranteed foolproof. They all require a cooperative programmer since there is no hardware prevention to keep the critical code sections from being executed without first checking the control bit. A possible solution might have been to use an immediate byte in the instruction as the control word, but since the IBR is used to buffer the instruction stream it is not

possible to keep other TP's from using the data while one TP is checking it. Another solution might have been to have had the instruction force an access, via some PR, using an address at some hardwired displacement from PR#0. This would not really solve the problem, however, since several pieces of code in various parts of memory may need to access the same data. In this type of case, a single, unique bit must be used for all sections. In actuality, we are trying to protect the data not the code which manipulates it. The only solution for a fool-proof lock-out system which would foil even the most malicious programmer would be to incorporate the locking out feature into the access mode protection. However, with the current hardware design there does not appear to be an easy way to do this.

One of the major contenders in the lockout problem was an instruction pair called Russian Roulette and Awake. Their operation is best explained through the use of an example. Consider the code represented by Figure 5.6.A.1/1 in which it is assumed that the section of code labelled A is always entered by executing the instruction labelled A.1 and left by executing the instruction labelled A.2.

Figure 5.6.A.1/1

The Russian Roulette instruction consists of a mnemonic byte followed by an operand phrase. The operand phrase is used to point to a byte containing the bit to be tested. When the instruction is decoded, the operand phrase is used to form an address. When the contents

of this location are fetched, the exchange net is not released. The bit is tested. If the bit is '0', the bit is set to '1' (written back to core) and execution continues with the next instruction in sequence. If the bit is '1', the exchange net is released and the TP kills itself (goes dormant). PRO is not advanced. (Thus arises the name of the instruction: if the bit is loaded, the TP kills itself.)

The AWAKE instruction at A.2 consists of a mnemonic byte followed by an operand phrase. Once again, the PR points to the same bit set at A.1. The bit at the specified address is set to '0' and a signal is sent to every TP to wake up. Those which are already running (including the TP which initiated the command) will ignore the command and continue their operation. Any TP's which are asleep, however, will immediately come into operation and again try to access the RR instruction which initially put it to sleep. (If several TP's try at once, the conflict will be resolved by the exchange net.) One TP will be allowed to access the region; the others in turn will eventually access the test bit which by this time will be on, so that they will again be killed.

One unpleasantness of the Russian Roulette-Awake scheme occurs when there are two critical sections, A and B, which are currently being executed by different TP's simultaneously. If a third TP now wishes to access region A, it will be killed. But suppose that in the interim section B finishes and the TP executing that section issues an awake instruction. This will cause the suspended TP to awaken. But since its section is still locked out, the TP will immediately be put back to sleep to await the completion of section A. Correct lock-out protection is achieved, but the mechanism here is less than optimum.

Two things should be noted about this pair of instructions. First of all note that if the bit at the test location is '1', the TP is suspended; it does not loop at this location a'la the TCOX instructions on the 7094 for instance. If this were to take place, an intolerable snarl might ensue due to conflicts at that memory, resulting in a

possible degradation in the execution speed by a factor of 2-4.

Secondly, it can be seen that each TP puts itself to sleep but can only be awakened by another TP. The process of being put to sleep or "killed" involves the suspension of all operations, probably including the timer.

After this instruction pair had been proposed, several other suggestions were made in regard to possible improvements. The original method might be called the 'dum-dum' method because it is explicitly assumed that each TP (IOP) knows nothing about the others.

In that method:

- 1) No record (queue) is kept of the order in which TP's attempt to access a critical section. The order is effectively built-in by the exchange net.
- 2) All dead TP's are awakened, not just those which wish to access this critical section.

The first suggestion is an attack on #2. Let the byte to be tested have the form:

The flag bit is used to determine whether or not the critical section may be entered. If so, set the flag = 1 and proceed. If not, TP_n sets bit n = 1 and dies.

Upon exiting from the section, the executing TP must reset the byte flag and then wake up a TP. To do this it examines the 4 bits in a wired-in order, say from left to right. The TP represented by the first non-zero bit is awakened and that bit is zeroed.

This formulation allows one to remove the necessity of awakening

all other TP's irrespective of whether or not they have tried to access this section and whether or not more than one TP wishes to access this section. Nevertheless, no mechanism is provided to queue the requests as they come in.

The price paid by the above seems large compared to the amount of improvement achieved. First, the instruction appears different to each TP. (A different bit must be set by each.) Secondly, each TP must have the capability of separately awakening all others. (This may not be too bad if handled by the Interrupt Unit, but it is an added complexity over the original Awake Instruction.)

The second proposal is an elaboration of the first. Here the test byte has the form:

Again, the flag is used to signal whether or not the critical section is in use.

$F_1 - F_3$ form a FIFO queue of depth 3, the depth being kept by the count in bits 6, 7. Thus, before killing itself after finding a section inaccessible, a TP must:

- 1) Set $cnt = cnt + 1$ (Initially 0)
- 2) Set $[F_{cnt}] = \# \text{ TP executing instruction } 0, \dots, 3$

(Obviously if $cnt = 3$ originally, we're hurting)

Upon exiting from the section (and resetting the flag)

1. if $cnt = 0$ do nothing; else
2. Wake up $TP_{[F_{(Cnt)}]}$
3. $Cnt = Cnt - 1$

This method allows one to maintain the order in which access attempts were made as well as needing to awaken only one specific TP (at a time). In this sense it has the best of all possible worlds. However, quite

a price must be paid in hardware.

The final argument which killed both of these supplementary suggestions, however, was the fact that very little efficiency is in fact wasted by making TP's which are asleep, wake up and then immediately go back to sleep again. All critical sections should, of course, be as short as possible. Therefore, no useful work could be accomplished in the time that the other TP's would otherwise spend sleeping. In both cases the added complexity to both the RR and the Awake instruction was exorbitant.

One major problem with the RR-Awake instruction pair was the need to use the Interrupt Unit to transmit the Awake signal. It could have been possible to have a special purpose bus between each of the TP's but it was felt that it would be better to centralize the transmission of all "interrupting type" operations through the Interrupt Unit.

It also turned out that the locking out feature of the Russian Roulette instruction (as distinct from the going to sleep feature) would be needed in other types of operations as well, in particular in the handling of interrupt storage (see Section 5.6.4.2). Thus it was eventually decided to have one general purpose instruction for checking a control word while locking out all other processors (INCK), and to use a separate instruction for putting a TP to sleep (SLEEP).

Secondly, it was decided that in order to avoid bothering the Interrupt Unit with waking up sleeping TP's, the sleep instruction would have a operand phrase-controlled time limit during which it would sleep. It's "alarm" would be controlled by the passage of the given amount of time on the TP's internal interval timer. This eliminated the need for the AWAKE instruction and simplified the IU logic slightly. However it also introduced the possibility that the TP might sleep too long after another TP had finished a critical section. This problem can probably be minimized by proper adjustment of the sleep period.

5.6.A.2 The Interrupt Storage Problem

One recurring problem in the Supervisor has been the problem of where to store the TP status information during an interrupt and what type of format to use within the interrupt storage area. It is clear that with the possible exception of a machine malfunction, each interrupt can and should be associated with a generating task (not processor). In order to handle the possibilities of multiprogramming and multiprocessing, the interrupt information must be deposited in an area addressed by the Segment Name Table (Local Segment portion) or within the Segment Name table itself. This isolates the machine states from each other and from dependence on any particular processor.

Unfortunately there are two mutually conflicting restrictions on the interrupt storage area.

- 1) It must be capable of being easily located by the hardware. This generally means that it should have a standard entry in the Segment Name Table.
- 2) Because the interrupt storage block is different for each task, it is technically a local segment. However if it were put in the local segment portion of the Segment Name Table its position would vary with the length of the number of entries in the global segment portion.

Several solutions to this conflict were suggested. One was to actually split the Segment Table into two parts. The segment names would have one bit which indicated to which part of the table they referred, the local or global portion. This would allow dynamic growth in both portions without readjusting the names already assigned. However, it would also require either two BR#0's or a sophisticated paging scheme for the Segment Name Table.

A second solution was to reserve a special area of the Segment Name Table specifically for interrupt storage. This would solve the problem of needing a local segment for interrupt storage since each

task obviously has a separate Segment Name Table. It would also make this area easy to access for the hardware since it could be put in a constant position in the Segment Name Table (e.g. the 1st page). However it adds the esthetic unpleasantness of having data in a table which was supposedly only for base descriptors.

Neither of these two solutions seemed entirely acceptable although the second one had fewer strong objections. However, before a final decision was made a second situation had to be considered.

Most tasks which run on the Illiac III system will not be big enough or important enough to justify parallel processing in which two processors will be working on different logical parts of the same task. Even if there were such a task, it is more than likely that in order to handle it the system would break it up into several tasks each of which would be handled separately and only need one processor. However there is one task in the system for which this is not true--namely the Supervisor Task. The Supervisor may very easily be running on more than one processor at a time performing operations for two different tasks.

Given such a situation a problem concerning interrupt storage arises. If two interrupts occur in two different TP's while the Supervisor is being run on both of them, both TP's may try to store interrupt information in the same area of core. Several possible solutions were suggested:

- 1) If the interrupt storage area is in the Segment Name Table use four different tables, one for each TP.
- 2) If the interrupt storage area is a segment, use 4 segments with 4 entries in the table, one for each TP.
- 3) Have the Supervisor's interrupt handler empty the interrupt storage block as soon as possible after an interrupt and store this data somewhere else in a stack. This data can then be processed later on, the basic idea being to clear the interrupt storage area as quickly as possible.

- 4) Use an interrupt storage area which makes use of circular buffer techniques or linked lists. This would allow information to be stored more permanently in the interrupt area and at the same time (assuming the interference between the accessing of links can be avoided) would allow almost simultaneous handling of interrupts on different TP's.

The first two solutions would involve some mechanism for either loading the BR#0 with a different base address for each TP which uses the Supervisor, or loading from a different entry in the Supervisor Segment Name table depending on which TP is being interrupted. Neither method was particularly pleasing esthetically since the Supervisor would become considerably different in structure than the other tasks.

The third solution did not really solve the problem since it would be hard to handle interrupts from one TP running the Supervisor while another TP is emptying the storage block. Here again the Supervisor would have considerably different operating procedures than the other tasks.

It was finally decided to use a circular buffer as in solution number 4. The details are described in Section 5.6.4.2. Linked lists were not used because no simple way could be found to restrict access to the links to one TP for the time it takes to modify the link structure. In a circular list since there is only one word which must be modified, the Exchange Net can simply be held open on that access until the modification is complete.

Two other reasons for using this scheme were:

- 1) Identical procedures can be used for all tasks whether or not they may engage in parallel processing without causing any extra overhead if they do not, and
- 2) The operations used to lock out other processors can be easily adapted for other parts of the system as well.

Finally, given this circular buffer format for the interrupt storage, it appears that the most desirable location for the interrupt storage buffer is in a separate segment with a normal entry in the Segment Name Table. In order to be accessed by core its entry is placed in the zeroth location in the Segment Table. As a result the Segment Name Table must be considered to contain the Interrupt Storage Segment entry as a special entry loaded prior to the Global and Local Segment areas.

5.6.A.3 Interrupt Handling

Perhaps the most difficult problem in Operating System design involves the handling of interrupts. As has been true of other large machines, interrupt handling is that single area which has precipitated the most problems and the most interaction with other parts of the Operating System design. This section describes the reasoning which led to the Illiac III Interrupt handling. To motivate the decisions made we describe earlier trial solutions which were discarded, their reason for exclusion, and the general evolution of the present system.

A fundamental policy question in the handling of interrupts (as in other areas of the Operating System) is where to place the hardware/software interface. The primary considerations were:

1. Who could do the needed operations most easily and efficiently?
2. Could a careless or malicious programmer disable the system if we allowed a given operation to be done in software?
3. Did we really know enough about the operational environment of the system to freeze the design in hardware?

The Original System:

In our first try at an interrupt system, we decided to divide the interrupts into various mutually exclusive levels in order to be able to reduce the reaction time and to simplify the software necessary to determine the interrupt which actually occurred. In addition we initially decided that when processing an interrupt at a given level no interrupts at a lower or equal level of priority would be recognized. These techniques are fairly standard on many machines and were considered to be a reasonable starting point for the interrupt design.

Given these conditions it was felt that an adequate interrupt handling system would involve separate storage areas for each interrupt level, into which would be stored the interrupt information and current status of the machine. The TP hardware was to recognize interrupts in reverse priority order (i.e. lowest priority first) so that if several interrupts occurred simultaneously, the lower priority ones would be saved first before work began on the highest one. The storage areas would be loaded such that when the interrupt at a higher level was finished, it would

"return" to start processing on the next lower priority interrupt.

Several problems developed with this scheme:

1. As the number of different types of interrupts expanded, it became problematical whether or not it would be possible to have a reasonable number of levels and still be able to guarantee that a given level would not contain two interrupts which under some set of conditions could occur simultaneously.
2. With the large number of possible interrupts it became very difficult to determine an optimal allocation of interrupt types to the various processing priority levels.
3. Many I/O interrupts may occur in a short interval of time; accordingly they may need to be stacked for later processing. This was not possible as long as equal priority interrupts could not interrupt each other.
4. "Supervisor type" interrupts also cause a variant of the above type of problem. For example, if in handling a call from a user program, the Supervisor exhausts one of its available space segments, it will need to request more storage from itself. Now, however, if one of the pages in the program which handles this is currently not in core (or if any one of several other interrupt possibilities occurs), there will be another interrupt--but this time the interrupt will not be recognized.
5. A multiprocessing problem occurs for the Supervisor task and those other tasks (if any) which are allowed to use concurrently more than one Taxicrinic Processor. If both TP's working on the task independently happen to detect interrupts at the same level within the same short interval of time, they might both try to store interrupt information in the same place, and chaos would ensue.

To a certain extent many of these problems could be handled by software. In particular, problem 4 could probably be handled by leaving certain pages in core at all times, checking the various Supervisor available space storage segments before they actually run out of space, and other similar safeguards to eliminate long series of "Supervisor type" interrupts.

This type fixup, however, is extremely time consuming and as the system becomes larger, it becomes almost impossible to keep track of all the details. As a result we have attempted to find a more general system which could eliminate some of these problems.

Circular Buffer Solution:

In an attempt to solve the multiprocessing problem(s) the Supervisor task's interrupt storage was expanded to allow for circular buffers in the place of the single storage block at each level. Each circular buffer consisted of several blocks for interrupt storage; the control of which block was to be used was invested in two control double words (CDW's). At first it was felt that only one circular buffer would be needed, instead of one for each level. But then it became apparent that if this were done, the first in-first out feature of the circular buffer would cause early low priority interrupts to be processed before later high priority interrupts. One possible solution would be to copy the information out of the circular buffer as soon as possible after the interrupt and to place it on an appropriate queue. However, this strategy would result in very inefficient use of the buffer and would waste much time.

The advantage of having a circular buffer for each interrupt level was that there was an automatic queuing of interrupts at each level of interrupt. In addition access to the control double words was restricted in such a way that two TP's could not simultaneously access the same storage block in the same interrupt level. As a result problems 3 and 5 were avoided.

To alleviate problems 1 and 4 using this scheme it was necessary 1) to allow interrupt processing at a given level to be interrupted by interrupts of equal priority as well as those of higher priority and 2) to place all those interrupts which require attention from the Supervisor in one level. This was still inadequate, however, since it is usually necessary to process the Supervisor type of interrupt on a last in-first out basis instead of the first in-first out basis which is used in the circular buffers.

The only way out of the situation, however, was to unload the circular buffer completely each time the Interrupt Handler Procedure was called and to stack the jobs appropriately. Once again, this solution would involve excessive data transfer due to the moderate size of the interrupt storage blocks. Thus a different procedure was needed to facilitate the

transfer of interrupt information.

Circular Pointer Buffer Solution:

The basic problem was now seen to be to find some form of storage from which the TP's could get room for interrupt information as it was needed and yet would lock out other TP's from the same storage. Circular buffers had the advantage of an easy lock out facility since the control double-words for the buffer were designed with this in mind. However, once data was in a circular buffer it became very difficult to manipulate it. Linked list storage structures were also considered, since they are ideally suited to the type of manipulation needed in this situation. Unfortunately, it is very difficult to lock out access to them since they can jump all over core.

The final solution was to use a circular buffer consisting of pointers to interrupt storage blocks. A circular buffer using this structure is shown in Figure 5.6.A.3/1. It retains the flexibility of a linked list structure since once the interrupt information has been written into a block it can be easily shifted from one queue to another. All that is necessary is to replace the empty position in the circular buffer of pointers with a pointer to a new unused block from available space whenever one removes a full block from the buffer. At the same time the circular buffer of pointers retains its lockout protection by use of the control double word.

Thus the proposed system can now take care of problems 1, 3, 4 and 5. As far as problem 2 is concerned, there does not appear to be an easy hardware solution. What we now have, however, is a very specialized system which is necessary in order for the Supervisor task to be able to handle interrupts but which may not usually be necessary for the vast number of other tasks in the system. Unfortunately since the Supervisor is so important, it is very necessary that these techniques be used to make it as efficient as possible. At the same time it is desirable that the other tasks not be saddled with superfluous garbage.

The final solution thus involved designing the combined hardware/software system so that each type of task could have as much of the full system as necessary and yet not suffer excessive inefficiency due to features it did not need.

Figure 5.6.A.3/1 Circular Buffer
of Storage Block Pointers

The Proposed System:

At the most complicated level, namely the Supervisor, the Interrupt Segment contains n circular buffers of pointer entries, one for each interrupt level. When an interrupt occurs, the TP uses a control double word to obtain an entry from the appropriate circular buffer and then it fills up the corresponding interrupt block. By loading the address of the circular buffer entry which was used into PR#1 before transferring control to the Interrupt Handler Procedure, there is no need to have a second control double word just to find out which interrupt must be handled next. The single CDW's for each level are stored in the beginning section of the Interrupt Storage Segment.

In those tasks and those interrupt levels which would never have multiple interrupts the circular buffer of pointers need only have one entry which points to a single interrupt storage block. Such a system involves two extra memory accesses over the original plan of a single block for each level. However this is not quite as inefficient as it seems since filling up an interrupt storage block already takes several double word accesses anyway.

It will be the responsibility of the Interrupt Handler Procedure to perform any necessary queueing operations before working on the interrupt itself. During this time all interrupts (except machine check?) will be inhibited.

Since all ordinary tasks use the same interrupt handling techniques they will be able to share a single Interrupt Handler Procedure. Only the Supervisor (and perhaps some other very special tasks) will need a special procedure.

Separation of Return Status and Interrupt Information:

At one point in the design procedure it was felt that the restriction on recognizing interrupts should be completely lifted so that the interrupts would be recognized as they occurred regardless of their processing priority level. This meant that high priority jobs would be interrupted "momentarily" to stack lower or equal priority jobs which would then be processed later. As it turned out, this change did not solve any problems and so it was abandoned. Nonetheless, some interesting points were discovered and the whole attempt tended to show

why other designs masked out lower priority interrupts during interrupt processing.

The problem which developed was that if one tries to stack a lower level interrupt while processing a high priority interrupt, a problem arose with the returning state of the machine. Since the processor was working on a high priority interrupt, the return state for the lower level interrupt will be somewhere in the program for processing the high priority interrupt. But since the high priority interrupt must be completed before the lower priority interrupt, by the time the low priority interrupt has been processed the return state for that interrupt will be invalid.

Lukily this problem led to the observation that in fact, the interrupt information which is saved upon the detection of an interrupt can be divided into two distinct parts:

1. that information which describes what interrupt took place and which provides data necessary to diagnose and/or correct the situation, and

2. that information which tells how to get back to the state of the machine just before the interrupt occurred.

These two parts are actually used at different times in the interrupt processing since part two is not needed until the processing has been completed while part one must be used before it can even start.¹

As a result of this observation it was felt that it might be desirable to store these two parts in separate areas of the interrupt storage segment. In particular it would seem desirable to continue to store the interrupt information (part 1) in an area associated with the

-
1. One of the advantages of using the two storage-area system for interrupts occurs in the case of I/O Interrupts. Usually when an I/O job sends an interrupt indicating that it has finished, it is more than likely that the task which initiated the job has been taken off of its processor and is in a wait state. This had always presented a conceptual difficulty since the task being interrupted in this case had absolutely nothing to do with the interrupt and yet it had to store the interrupt in its own interrupt storage block and treat the interrupt as one of its own. The new system did not basically change this problem but it does present a new way of looking at it, since the interrupt information has been split up into that belonging to the task waiting for the I/O completion and that belonging to the task which was interrupted.

level of the detected interrupt. However, the return status information (part 2) should be stored in an area associated with the level of interrupt operation being interrupted. (Level 0 could refer to the task itself and levels 1 to n to the various interrupt priority levels). This system would then allow the interrupt handler to return to processing an interrupt at a particular level simply by loading the current status contained in the return status block for that particular level.

In this scheme the problem of knowing what level was being interrupted was to be solved by having an interrupt level register within each TP and having the TP hardware control every transfer between levels so that there was no way to get from one level to another without the hardware knowing about it and adjusting the interrupt level register accordingly. The problem of initializing the return status information could be handled by the system by initially loading the blocks so that they would cause a return to the appropriate level of processing. Then by allowing the TP hardware to be the only means of changing and using the return status information, the status would hopefully remain in a valid state for each level from then on.

As was previously mentioned however, the system still did not help to straighten out the confusing mess about which interrupts to handle first. Also the hardware for the new system seemed to be much more complex and it was not really obvious that it was stable enough to not need changes in the algorithm when the system eventually ran. As a result it was decided to drop this aspect of the design.