

OLD EDITION

SITE COPY = CONTROL DATA ENGINEERS

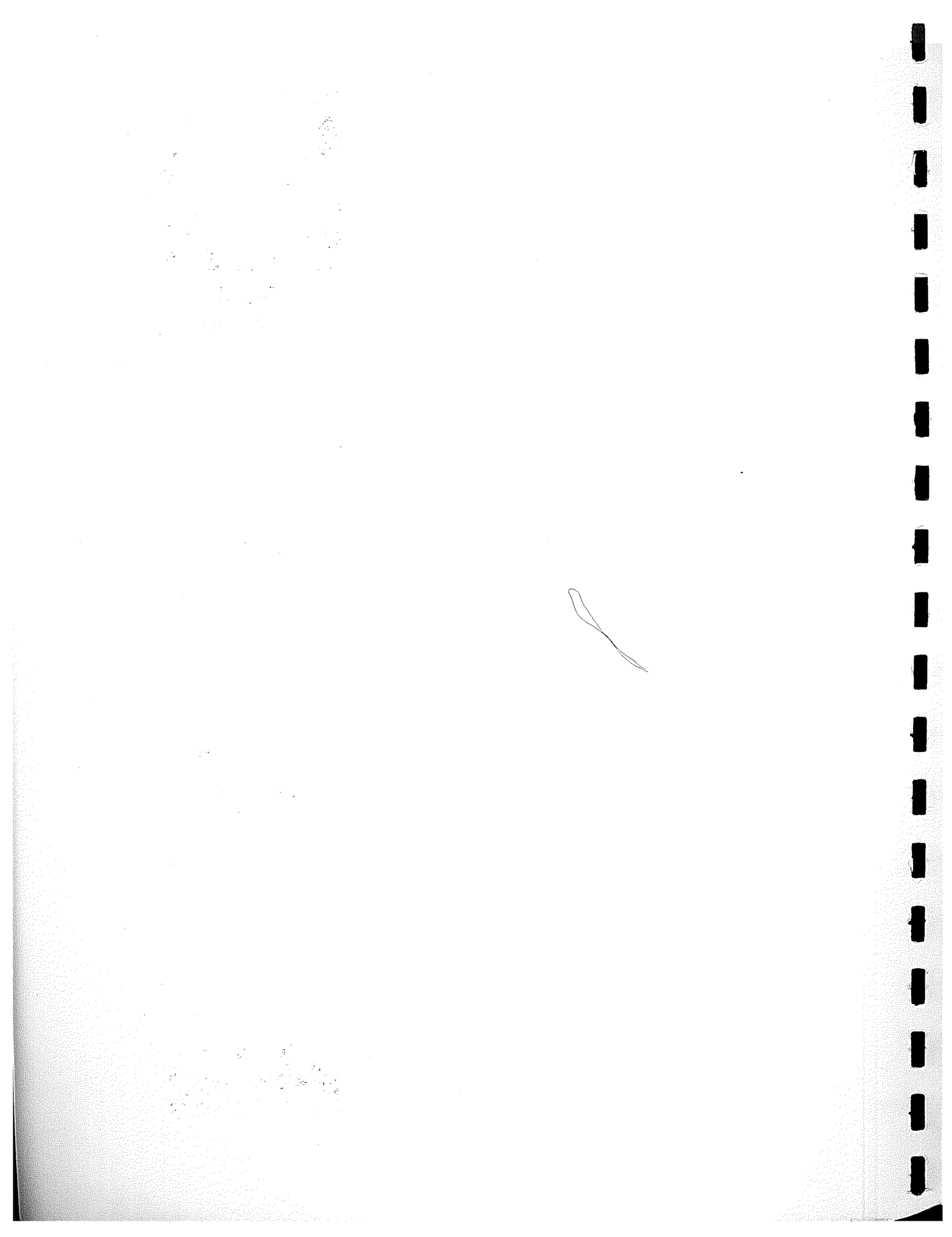
UNIVERSITY OF ADELAIDE



VOLUME II

PRELIMINARY
EDITION

CONTROL DATA
CORPORATION



SECTION 7.2

SHIFT

Functional Units



SHIFT FUNCTIONAL UNIT

7.2.1 INTRODUCTION

The Shift Functional Unit performs shift, normalize, round, pack, unpack, and mask operations as required by instructions 20 through 27 and 43.

The functional unit time is 400 nanoseconds for the normalize instructions (24 and 25) and 300 nanoseconds for any of the other shift operations.

The time difference arises because during normalize, a shift count must be generated by the Normalize Network before shifting of the coefficient takes place. The breakdown of the functional unit time into Read, Execute, and Store cycles will then differ as follows:

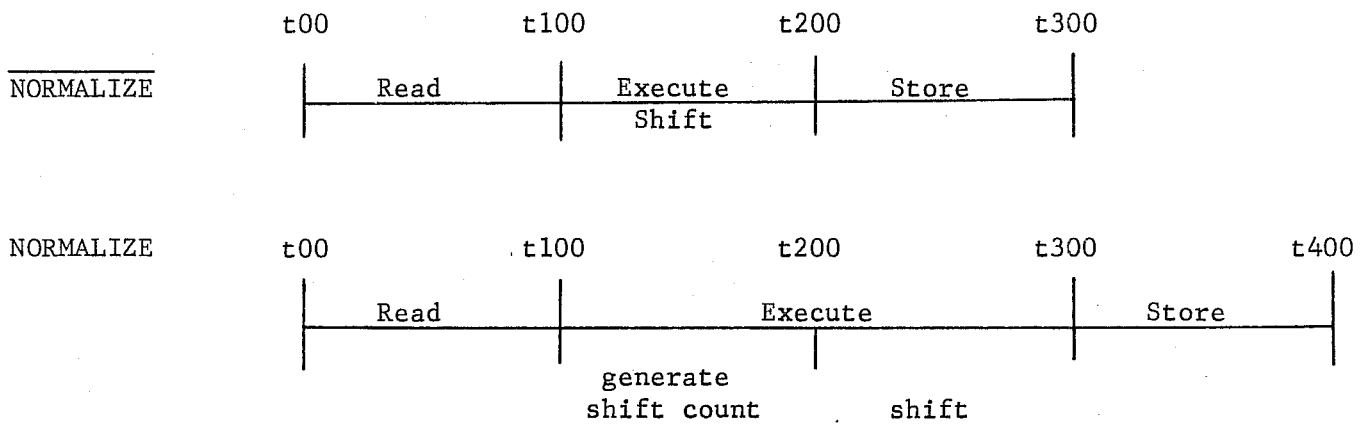


FIGURE 7.2.1

The Shift Unit shares data trunk number 1 with the Add and Long Add Units. It holds second priority on the Read Operand trunk and first priority on the Result trunk. The Unit may select operands from registers Xk, Xi, or Bj and may designate a result register of Xi and/or Bj. Also, the six bits, jk, may be used to specify a shift count - these bits are unconditionally sent to the Shift Unit each minor cycle and are used only if required by the instruction (used by instructions 20, 21, or 43).

The Shift Unit follows the functional unit principles discussed in the Boolean chapter (Section 7.1) in that it uses mode bits and a timing chain to select and sequence the required operations. In the standard manner, it is initiated with a "Go" signal from the scoreboard and terminates its operation (Clears its Busy flipflop and transmits its result(s)) upon being released by the scoreboard. It is on the other hand, a somewhat more complex unit because more intricate and varied functions are defined by its instruction list.

The main component of the Shift Unit is of course the Shifting Network, which is a 60-bit, 6-rank shifter that operates in nearly the same manner as the Peripheral Processor shift network. The six ranks enable shifts of 1, 2, 4, 8, 16, or 32 places in either the left or right direction. Shift direction and magnitude are determined by two circuits, Shift Direction Control and the Shift Count Register (SK). Shift Direction Control will determine and enable shift direction (left or right) by checking the mode bits of the Shift Unit.

The Shift Count Register contains six-bits each of which conditions one rank of the shift network. (Bit 2^5 conditions the 32 place shift rank, bit 2^4 the 16 place rank, etc.) A rank is enabled if its corresponding bit in SK is set, and disabled if that bit is a zero. The maximum shift count is thus $77_{(8)}$ or $63_{(10)}$ places (when all bits of SK are set.).

The shift count may come from any one of three sources, depending upon the instruction being executed; from 1) the six bits, jk, 2) the lower six bits of Bj (during nominal shifts), or 3) the Normalize Network which generates the shift count required to normalize a given coefficient. Another component of the Shift Unit to be discussed is the Bj Ones Test Network. During Nominal Right shifts, this network looks for any "one" in bit positions 6 through 10 of

Bj. If a one is found, the result of the shift network is not enabled to the shift network; the result is thus an all zero coefficient. To understand the reason for this circuit, consider that if any one of bits 6 through 10 is set, a right shift should result in an all zero coefficient (right shifts are by nature "end-off" and the shift count is greater than 63_{10}). If the network was absent, an erroneous, non-zero coefficient would be generated for all cases where Bj bits 0 through 5 gave a magnitude of less than 64_8 (since these six bits would unconditionally be used as the shift count). The Ones Test Network then guarantees an all zero coefficient for the case illustrated. The final component of the shift unit to be discussed is the Exponent Adder. It is used during Normalize to subtract the normalized shift count from the original operand (X_k) exponent. This insures that the normalized number is of the same value as the original. Further analysis of the components mentioned here will be found on the later pages of this section. First, the Shift Unit instructions and data flow are discussed.

7.2.2 INSTRUCTION LIST

Data paths for the following instructions may be seen by referring to Block Diagram #1, Figure 7.2-2. The expressions in parenthesis following the instruction names are the ASCENT symbolic codes.

20 SHIFT Xi LEFT jk places (LXi jk)

DEFINITION: This instruction shifts the 60-bit word in X register i left circular jk places. The 6-bit shift count, jk, allows a complete circular left shift of X register i.

DATA FLOW: The X Input Register is transferred to the Shift Input Register (SIR) and the jk Input Register to the Shift Count Register (SK). The gates for the shift network are controlled by the SK register and the shift direction translation. The output of the shift network is gated directly to the chassis line drivers.

21 SHIFT Xi RIGHT jk places (AXi jk)

DEFINITION: This instruction shifts the 60-bit word in X register i right jk places. The shift is end-off with sign extension.

DATA FLOW: Same as the 20 instruction.

SHIFT FUNCTIONAL UNIT BLOCK DIAGRAM #1
 (for instructions 20, 21, 22, 23, or 43)

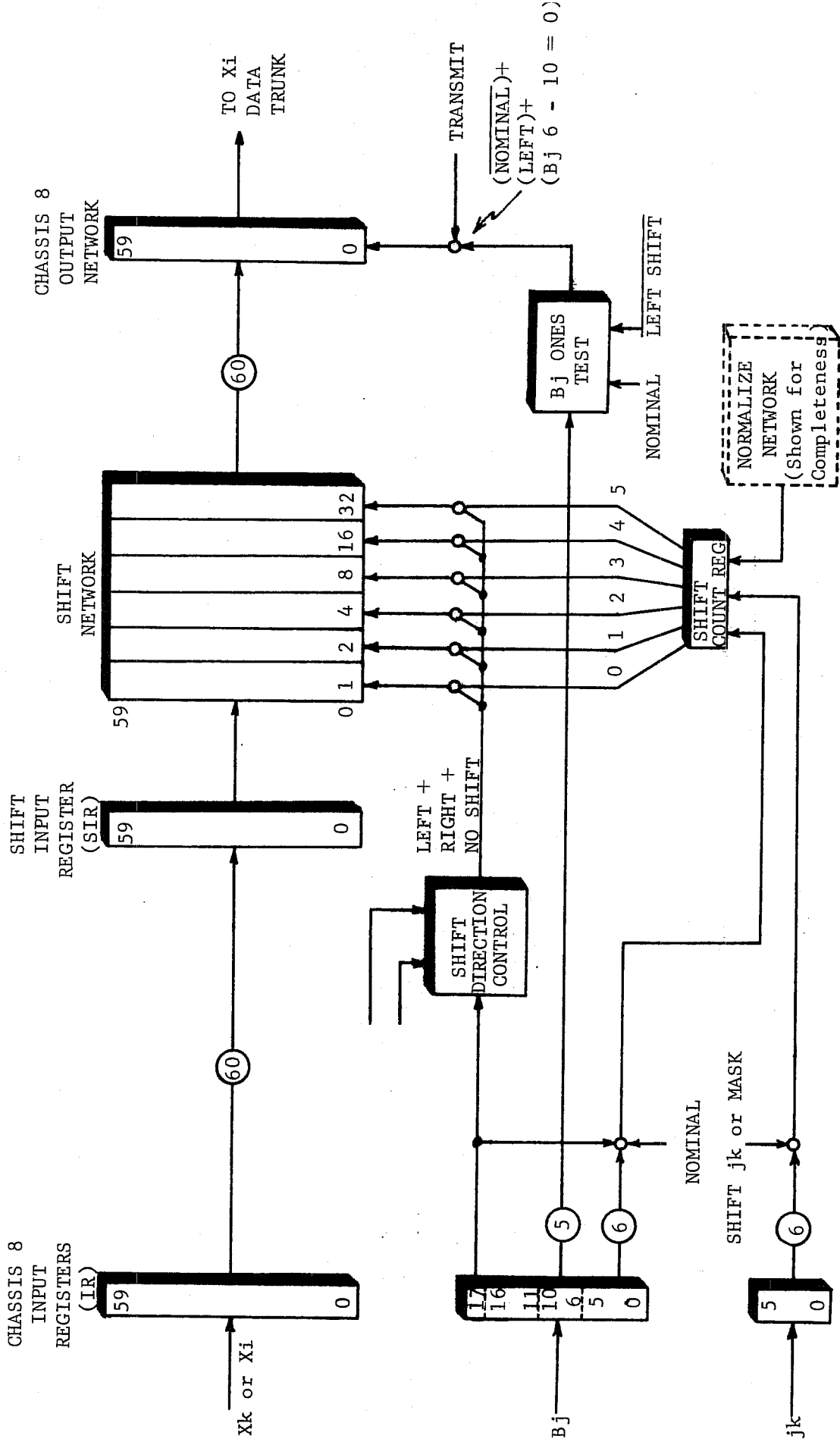


FIGURE 7.2-2

22 SHIFT X_k NOMINALLY LEFT B_j places to X_i ($LX_i = B_j, X_k$)

DEFINITION: This instruction shifts the 60-bit word in X register k the number of places specified by the low-order six bits (0 - 5) of B register j and places the result in X register i. If B_j sign (bit 17) is positive, the shift is left circular; if B_j sign is negative, the shift is right (end-off with sign extension).*

DATA FLOW: The X Input Register is transferred to SIR and the lower six-bits of B_j Input Register to SK (complemented if B register j sign is negative). The gates for the shift network are controlled by the SK register and the shift direction translation. The specified shift direction is reversed if B register j sign is negative. The output of the shift network is gated directly to the chassis line drivers.

23 SHIFT X_k NOMINALLY RIGHT B_j places to X_i ($AX_i = B_j, X_k$)

DEFINITION: This instruction shifts the 60-bit word in X register k the number of places specified by the low-order six bits of B register j and places the result in X register i. If B_j sign (bit 17) is positive, the shift is right (end-off with sign extension);* if B_j sign is negative, the shift is left circular.

DATA FLOW: Same as the 22 instruction.

Data paths for the following instructions may be seen by referring to Block Diagram #2, Figure 7.2-3

*The B_j Ones Test Network checks bits 6-10 of B register j during 22 and 23 instructions. If any bit is a one during nominal right shifts, all zeros are sent to X_i .

SHIFT FUNCTIONAL UNIT BLOCK DIAGRAM #2
 (for instructions 24 and 25)

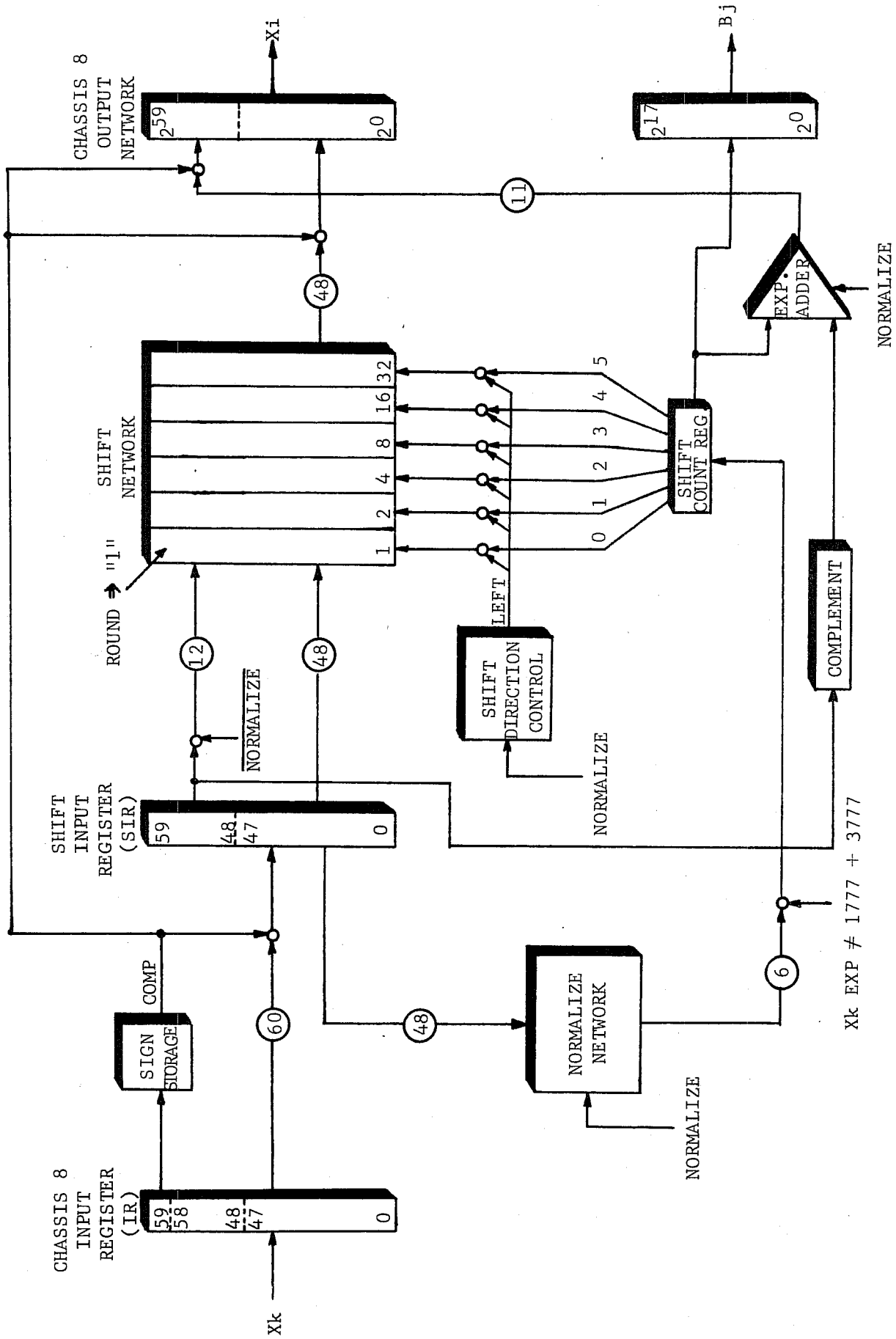


FIGURE 7.2-3

24 NORMALIZE X_k in X_i and B_j ($NX_i, B_j = X_k$)

DEFINITION: This instruction normalizes the floating-point quantity in X register k and places it in X register i. The number of shifts required to normalize the quantity is entered in B register j. A normalize operation may cause underflow, in which case both exponent and coefficient will be cleared; the normalize count is still entered in B register j. Normalizing a zero coefficient reduces the exponent by 48_{10} (60_8). If X_k is in infinite or indefinite form, it is sent out in tact and the normalize count is sent out as zero.

DATA FLOW: The X register sign bit is stored in a flip-flop to control data flow, and the X Input Register is transferred to SIR (complemented if X_k sign is negative). Bits 0 through 47 feed the normalize network which determines the number of zeros from bit 47 to the left-most "1" of the coefficient. The output of the normalize network (the normalize shift count) is gated to the SK register which, with the shift direction translation (always LEFT during normalize operations), controls the gates for the shift network. The transfer of the normalize network to SK is disabled if X_k exponent equals 1777 or 3777. The output of bits 0 through 47 of the shift network are gated (complemented if X_k sign was negative) to the chassis line drivers. The complement of the exponent portion of SIR and the true value of SK feed the exponent adder where the normalize shift count is subtracted from the exponent portion of SIR. The difference is the exponent portion of the normalized number and is gated directly to the chassis line drivers.

25 ROUND and NORMALIZE X_k in X_i and B_j ($ZX_i, B_j = X_k$)

DEFINITION: This instruction performs the same operation as instruction 24 except that the quantity in X register k is rounded by $\frac{1}{2}$ if that quantity is shifted. (It would not be shifted if the original quantity was already normalized.) A normalize operation may cause underflow in which case both exponent and coefficient will be cleared. Normalizing a zero coefficient places the round bit in bit 47 and reduces the exponent by 48_{10} (60_8). If X_k is in infinite or indefinite form, it is sent out in tact and the normalize count is sent out as zero.

DATA FLOW: Data paths are the same as the 24 instruction with the addition of the round operation. With "Round" specified, a one bit is forced in position 59 of the shift network. As in the 24 instruction, LEFT shift is specified. If the coefficient is shifted, the round bit will be pulled around below the least significant bit, thus adding a $\frac{1}{2}$ round bit.

Data paths for the following instruction may be seen by referring to Block Diagram #3, Figure 7.2-4.

26 UNPACK X_k to X_i and B_j ($UX_i, B_j = X_k$)

DEFINITION: This instruction unpacks the floating point quantity in X register k and sends the 48-bit coefficient with sign extended in the upper 12-bits to X register i . The 10-bit exponent (unbiased, sign extended, and represented in true one's complement) is sent to B register j .

DATA FLOW: The X register sign bit is stored in a flip-flop to control data flow and the X Input Register is transferred to SIR. Bits 0-47 are sent directly to Bits 0-47 of the X_i output network. The coefficient sign is extended to bits 48-59 through a fan-out. Bits 48-57 of SIR

SHIFT FUNCTIONAL UNIT BLOCK DIAGRAM #3
(for instruction 26)

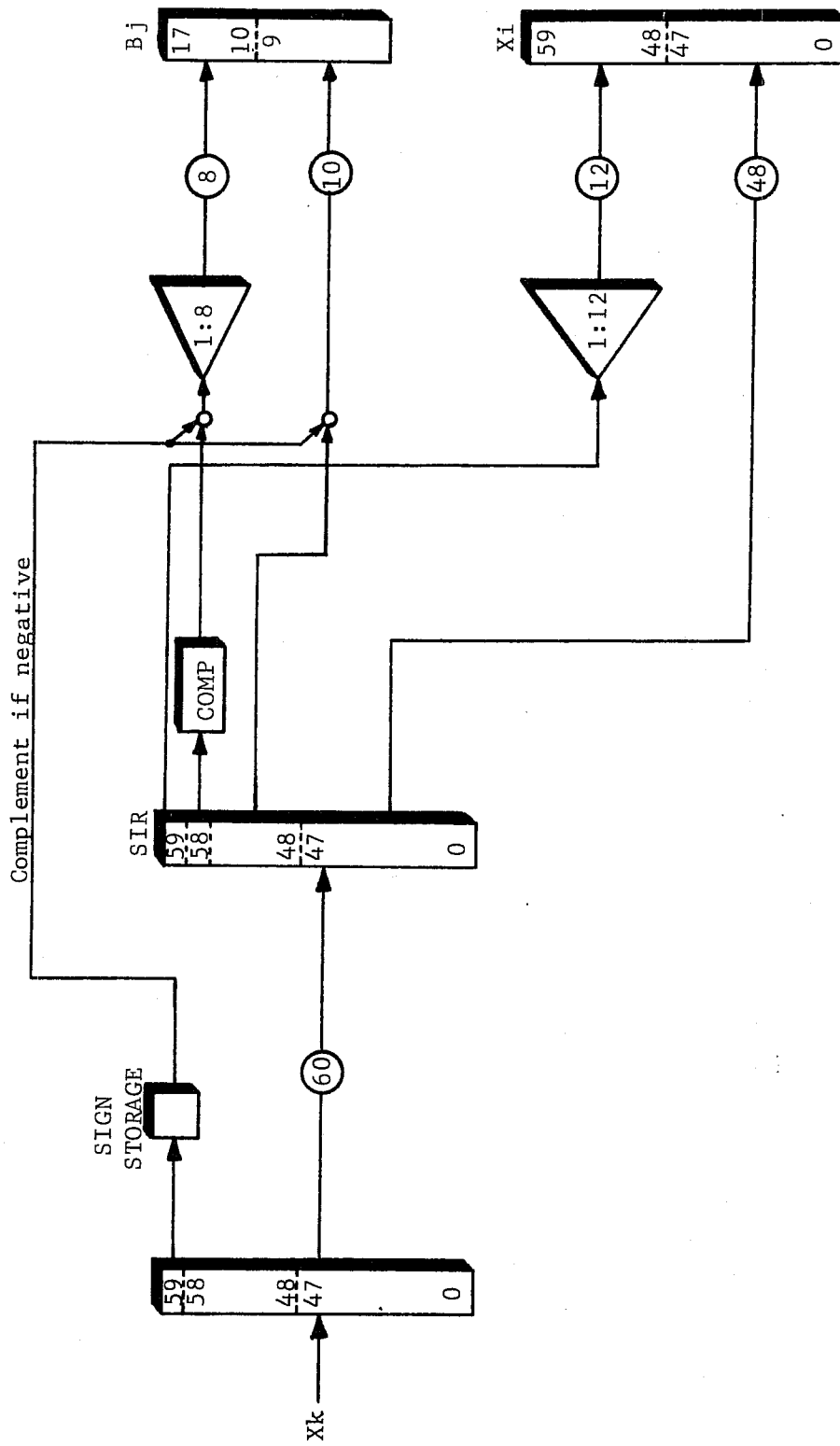


FIGURE 7.2-4

are sent directly to bits 0-9 of the Bj output network - complemented if Xi sign (bit 59) is negative. To remove the exponent bias and provide proper sign extension, the complement of SIR bit 58 is fanned-out to bits 10-17 of the Bj output network. These 18 bits will be complemented at the Bj output network if Xk sign (bit 59) is negative.

Data paths for the following instruction may be seen by referring to Block Diagram #4, Figure 7.2-5.

27 PACK Xi from Xk and Bj ($PX_i = B_j, X_k$)

DEFINITION: This instruction packs a floating point quantity in X register i. The coefficient is obtained from the lower 48 bits of X register k and the exponent from the lower 10 bits of B register j.

Bias is added to the exponent during the pack operation.

DATA FLOW: The X register sign bit is stored in a flip-flop to control data flow and bits 0-47 of the X input Register are transferred to bits 0-47 of SIR. Bits 0 through 10 of B register j are transferred to bits 48-58 of SIR. The setting of SIR bit 59 is disabled during PACK operations. The word now assembled in SIR is gated to the Xi data trunk. SIR bits 0-47 are transferred in true form. Bits 48-59 (bit 58 is complemented out of SIR to remove bias and bit 59 was made a "zero") are transferred in true form if Xk sign is positive; in complement form if Xk sign is negative.

Data paths for the following instruction may be seen by referring to the block diagram on page .

SHIFT FUNCTIONAL UNIT BLOCK DIAGRAM #4
(for instruction 27)

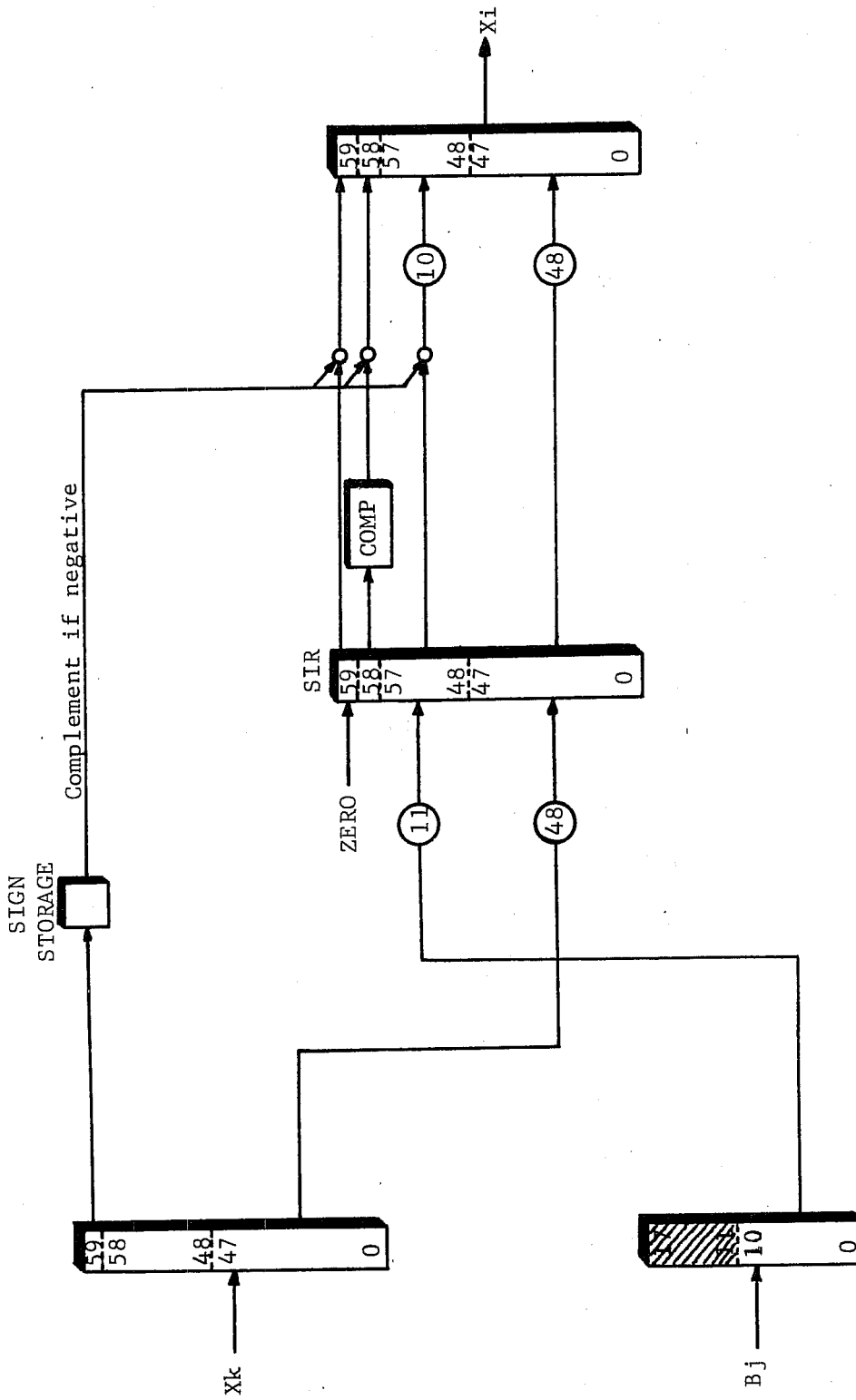


FIGURE 7.2-5

43 FORM jk MASK in X_i (MX_i jk)

DEFINITION: This instruction forms a mask in X register i . The 6-bit quantity jk defines the number of ones in the mask as counted from the highest order bit in X register i . If jk equals zero, X register i will equal all zeros.

DATA FLOW: The SIR is cleared, the jk count is transferred to the SK register, and a right shift is translated. The mask mode will cause a negative sign to be extended during the right shift. The overall effect is that an all zero operand is right shifted jk places with ones forced as sign extension. Thus, a mask jk places long is formed at the output of the shift network. This output is gated to the X_i line drivers.

7.2.3 MODE BITS

The following chart, Figure 7.2-6, summarizes the nine instructions that use the Shift Unit.

CODE	NAME	SHIFT COUNT	SOURCE	RESULT	TIME (NSEC)
20	Shift Left	jk	X_i	X_i	300
21	Shift Right	jk	X_i	X_i	300
22	Shift Left Nominally	$B_j(\text{bits } 17, 5-0)$	X_k	X_i	300
23	Shift Right Nominally	$B_j(\text{bits } 17, 5-0)$	X_k	X_i	300
24	Normalize	Normalize Network	X_k	X_i and B_j	400
25	Round and Normalize	Normalize Network	X_k	X_i and B_j	400
26	Unpack	None	X_k	X_i and B_j	300
27	Pack	None	X_k and B_j	X_i	300
43	Mask	jk	None	X_i	300

FIGURE 7.2-6

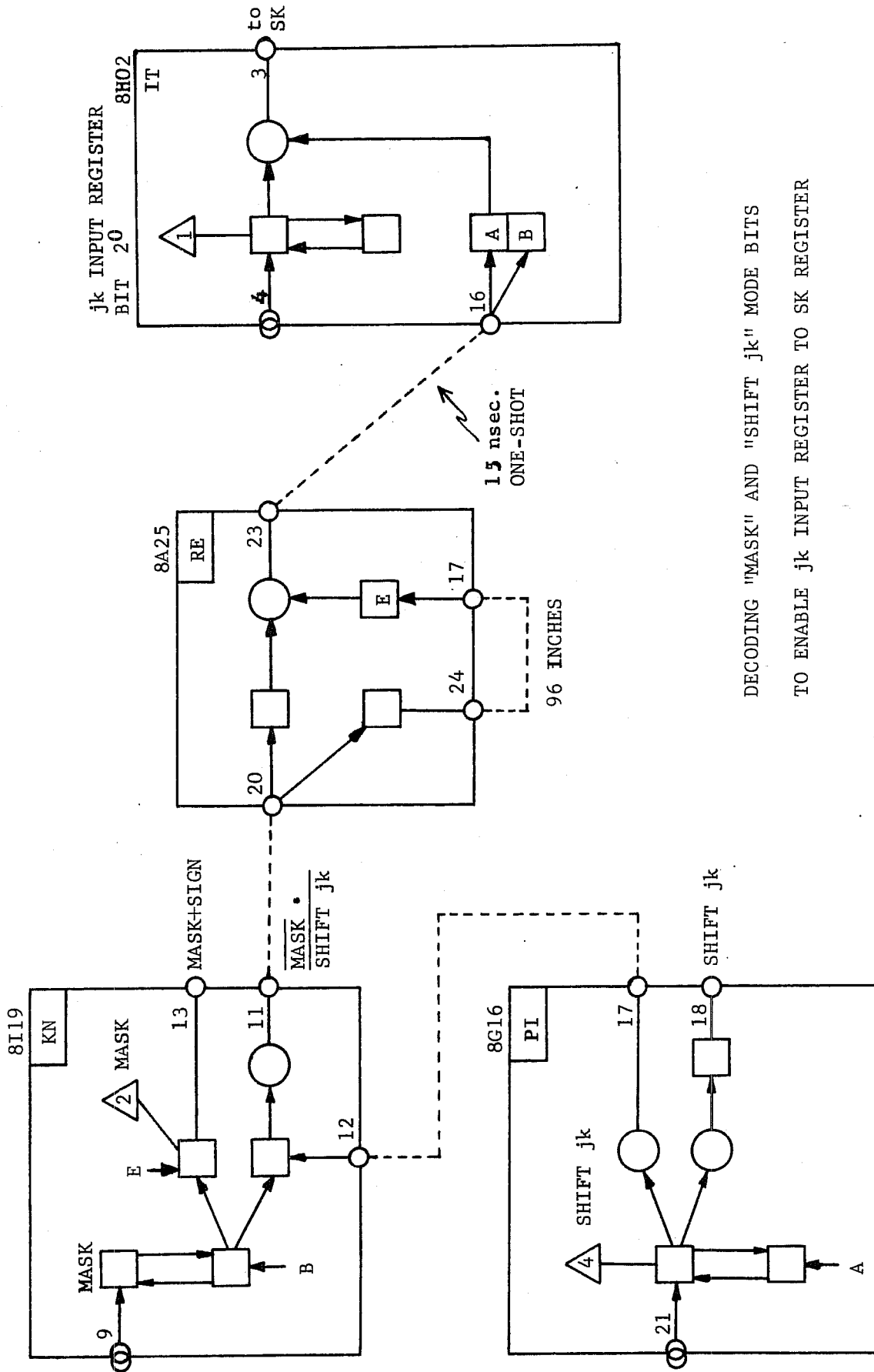
Note that two instructions (20 & 21) specify Shift jk, two (20 & 22) specify Shift Left, two (21 & 23) specify Shift Right, two (22 & 23) are Nominal Shifts, two (24 & 25) are Normalize instructions, and there is one each of Mask (43), Round (25), Pack (27), and Unpack (26). With this information, the following list of Mode Bits used by the Shift Functional Unit can be derived.

<u>MODE BITS</u>	<u>fm (INSTRUCTION)</u>
Shift jk	20 & 21
Shift Left	20 & 22
Shift Nominal	22 & 23
Mask	43
Normalize	24 & 25
Round	25
Pack	27
Unpack	26

The Mode Bits are translated in the same manner as the Boolean mode bits were, that is, the bits are ANDed, ORed, and fanned out to enable the various operations required by each instruction. Figure 7.2-8, for example, shows the decoding of the "Mask" and "Shift jk" mode bits to enable the six bits, jk, to the Shift Count Register (SK) during instructions 20, 21, and 43. The Mode Bit translators, transmitters, and receivers are shown on sheet 110 of the Shift Functional Unit Customer Engineering Diagrams. Complete translations may be made by referring to the Chassis 8 Wiring Tabs. Figure 7.2-7 summarizes the Mode Bits and the associated instructions.

Code	Name	Shift jk	Left Shift	Shift Nom	Mask	Norm.	Rnd.	Pack	Unpack
20	Shift Left	X	X						
21	Shift Right	X							
22	Shift Left Nominally		X	X					
23	Shift Right Nominally			X					
24	Normalize					X			
25	Round and Normalize					X	X		
26	Unpack								X
27	Pack							X	
43	Mask				X				

FIGURE 7.2-7



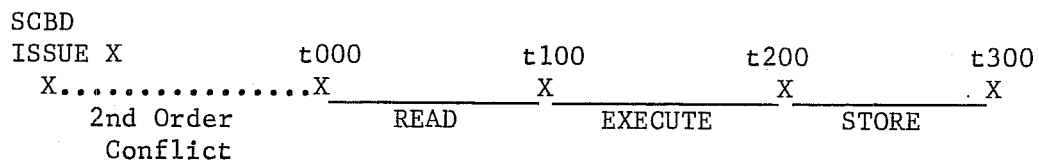
DECODING "MASK" AND "SHIFT jk" MODE BITS
 TO ENABLE jk INPUT REGISTER TO SK REGISTER

FIGURE 7.2-8

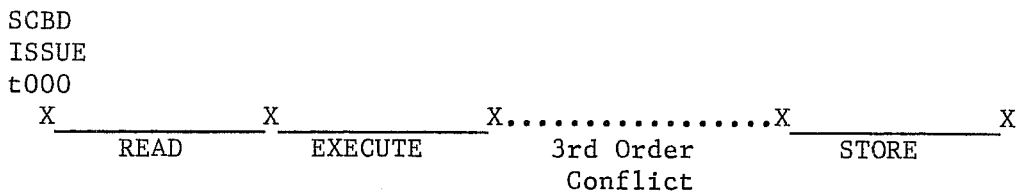
7.2.4 TIMING CHAIN

As was mentioned in the introduction to this section, (7.2.1) the Shift Unit time duration is 100 nanoseconds longer for Normalize than for Non-Normalize shift class instructions. This occurs because during Normalize, a Shift count must be generated by the Normalize Network of the Shift Unit while for other Shift instructions, the shift count is available at the same time as the operand.

The timing sequence for the Shift unit is shown in Figure 7.2-9 and an explanation of the timing is found on the facing page. The timing sequence assumes that no second or third order conflicts occur. If a second order conflict does occur, the "Go Shift" pulse will be delayed for the duration of the conflict.

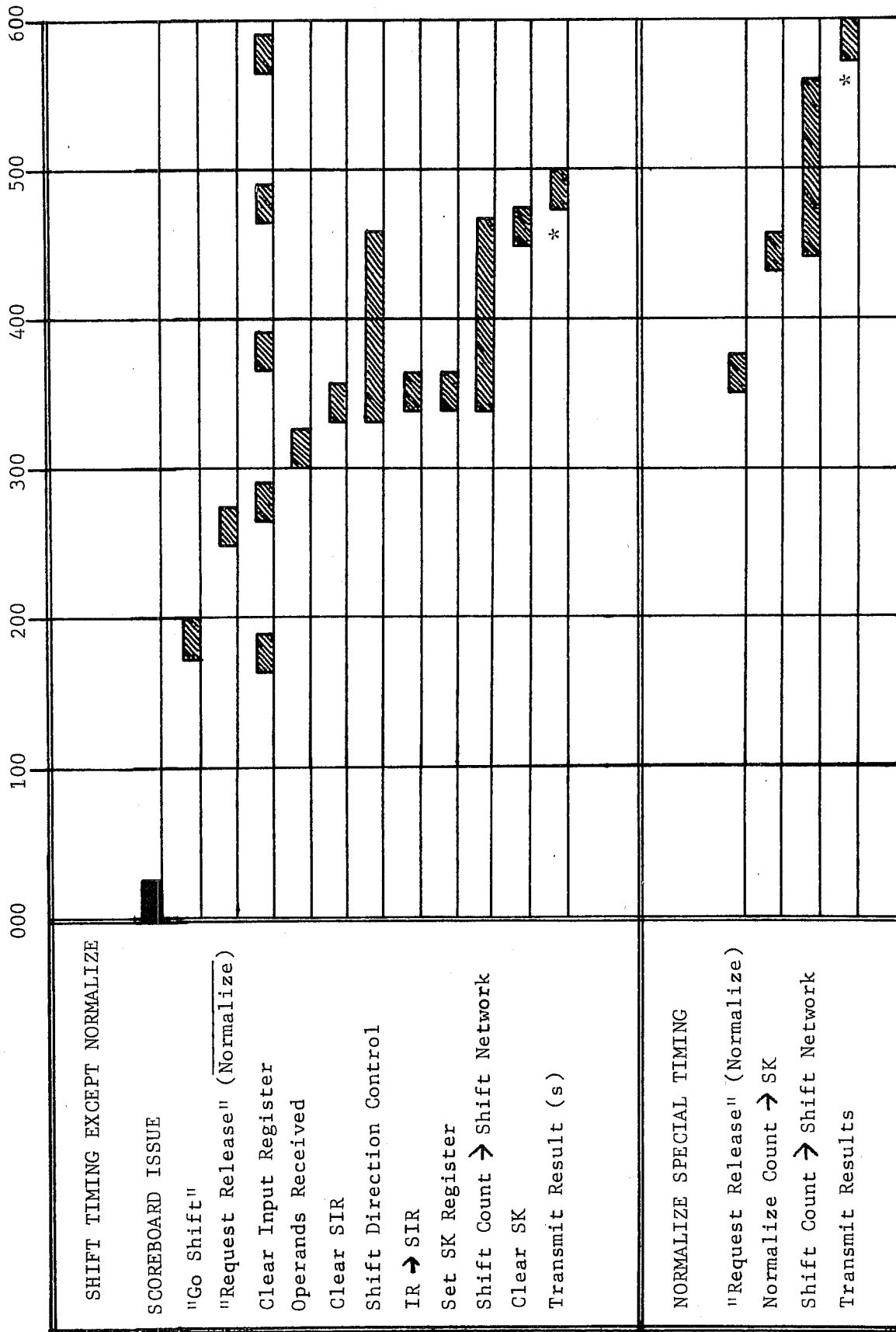


If a third order conflict should occur, gating of the result to the register chassis (Transmit Result) will be delayed for the length of the conflict.



Thus, 300 nanoseconds is the Functional Unit time for Non-Normalize instructions; 400 nanoseconds for the Normalize and the Round and Normalize instructions.

SHIFT FUNCTIONAL UNIT TIMING CHART



* Earliest possible time - no result register conflict.

FIGURE 7.2 9

SHIFT TIMING (EXCEPT NORMALIZE)

- t000 - Issue of the Shift Instruction to the Scoreboard
- t025 - The jk portion of the current Shift instruction is received at the Shift Functional Unit. (jk is unconditionally sent to the shift unit every 100 nanoseconds, and is gated to SK upon receipt of a "Mask" or "Shift jk" mode bit) (See Figure 7.2-8)
- t100 - If "Shift jk" or "Mask" mode bit was received, the jk catching register is transferred to the SK register.
- t175 - The "Go Shift" pulse starts the Shift Unit timing chain
- t250 - The "Request Release" pulse is sent to the All Clear Network (SCBD) if the Normalize mode is NOT specified.
- t300 - Source operands are received at the Input Registers of Chassis 8.
- t325 - The Shift Input Register (SIR) is cleared in preparation for the receipt of the operand.
- t330 - Shift Direction Control is enabled to the shift magnitude (SK) AND gates which will in turn enable the Shift Network.
to
t460
- t340 - (1) Source operands are gated from the Input Registers to SIR. (2) Bj (if Nominal Mode) shift count is gated to the SK register.
- t340 - Shift magnitude AND gates enable the Shift Network. Approximately 130 nanoseconds are allowed for filter time through the static
to
t470 shift network.
- t400 - The "Transmit" pulse is received at the Shift Unit (assuming no third order conflicts exist)
- t450 - The SK register is cleared as a result of the receipt of the "Transmit" pulse.
- t475 - The Shift Unit result is sent to the register chassis, gated by "Transmit", and will be received at Entry Control at about t500.

NORMALIZE SPECIAL TIMING

- t350 - The "Request Release" pulse is sent to the All Clear Network if the Normalize mode is specified.
- t430 - The output of the Normalize Network (normalize shift count) is gated to the SK register.
- t440 - The Shift Magnitude AND gates enable the Shift Network. Approximately 130 nanoseconds are allowed for filter time through the
to
t570 static shift network.
- t500 - The "Transmit" pulse is received at the Shift Unit.
- t550 - The Shift Count Register is cleared due to the receipt of the "Transmit" pulse.
- t575 - The Shift Unit result is sent to the register chassis, gated by "Transmit", and will be received at Entry Control at about t600

Reference is made to the Shift Functional Unit Customer Engineering Diagrams, Sheet 111, where the logic associated with Shift timing can be seen.

7.2.5 SHIFT DIRECTION CONTROL

As is implied by the name, the function of Shift Direction Control is to determine the direction (left or right) of shift for the Shift class instructions. Mode bits and the negative or positive condition of Bj sign are logically combined on a CT module (I17) whose output will ultimately specify Left or Right direction. The direction is combined with bits from the Shift Count Register (SK) on CA modules which are then fanned out to enable the six ranks of the shift network. One of three possible enables will condition rank "X" of the Shift Network (where X = 1, 2, 4, 8, 16, or 32).

1. Shift Left "X" places (if Direction Control \implies Left)
2. Shift Right "X" places (if Direction Control \implies Right)
3. No Shift "X" places (if the SK bit for magnitude "X" = 0)

During the following explanations, refer to Figure 7.2-10, a logic drawing of Direction Control.

Left shifts are possible only with the following instructions and conditions:

- 20 This instruction specifies an unconditional LEFT shift in the shift constant (jk) mode. Mode bits "Shift jk" and "Shift Left" are ANDed (I17, inverter L) and force a "one" out of test points one and two and pin 9.

SHIFT DIRECTION CONTROL

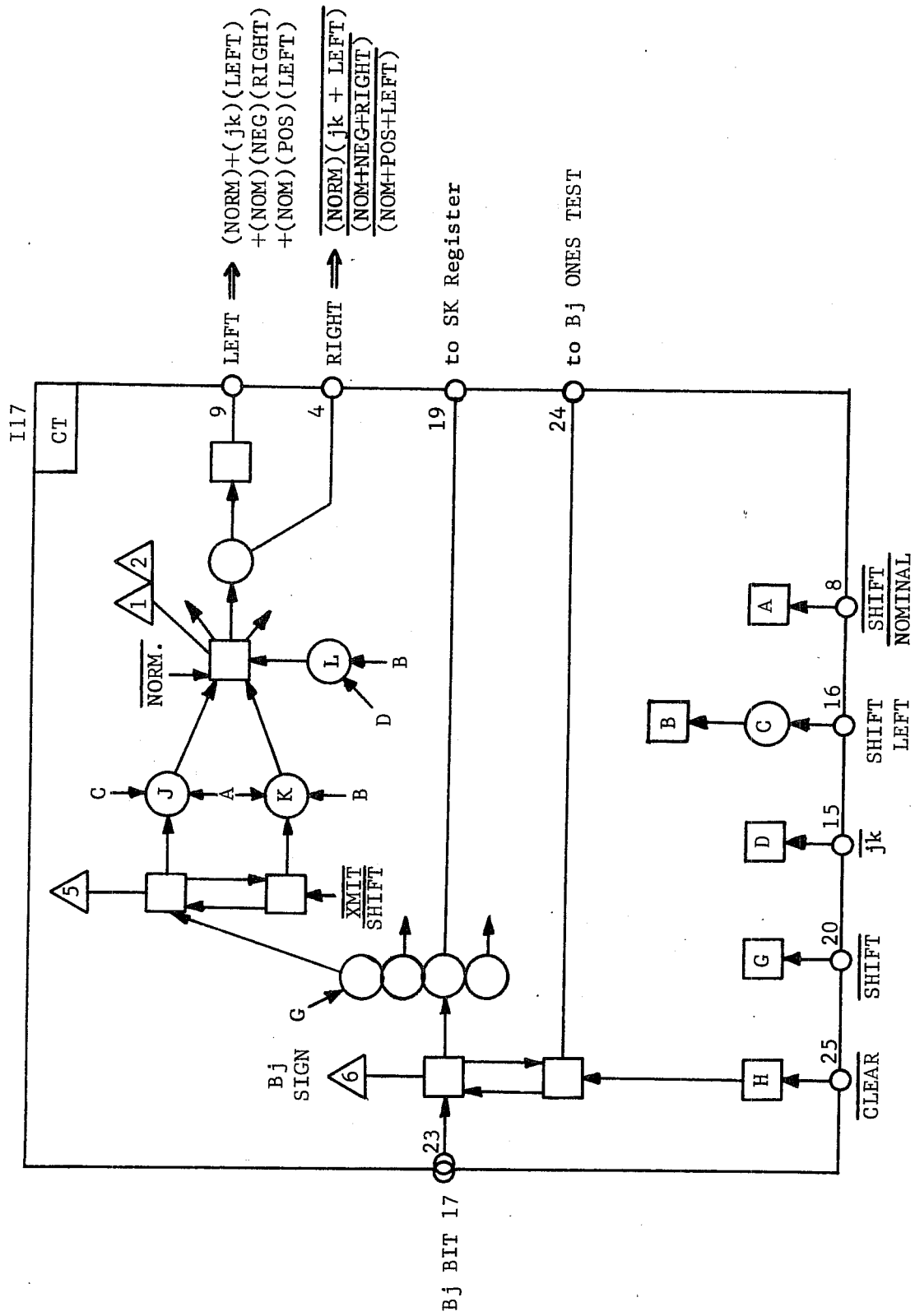


FIGURE 7.2-10

- 22 This instruction specifies a nominal LEFT shift if the sign at B register j is positive. (if Bj sign is negative, the shift direction will be right.) Mode bits "Nominal" and "Shift Left" and the condition "Bj is positive" are ANDED (I17, inverter K) and force a "one" out of test points one and two and pin 9.
- 23 This instruction specifies a nominal right shift if the sign of B register j is positive. If Bj sign is negative, the shift direction will be LEFT. Mode bits "Nominal" and "NOT Left Shift" and the condition "Bj is Negative" are ANDED (I17, inverter J) and force a "one" out of test points one and two and pin 9.
- 24 and 25 Both of these instructions specify the Normalize mode of operation. A LEFT shift is always required during Normalize; therefore, the "Normalize" mode bit forces a "one" out of test points one and two and pin 9 of module I17.

Right shifts are possible only with the following instructions and conditions. A right shift will result in "zeros" at the outputs of test points one and two and a one out of pin 4 since all three AND gates (J, K, & L) and the Normalize condition will be "ones."

- 21 This instruction specifies an unconditional RIGHT shift in the shift constant (jk) mode.
- 22 This instruction specifies a nominal left shift if B register j sign is positive. If Bj sign is negative, the shift will be RIGHT.
- 23 This instruction specifies a nominal RIGHT shift if B register j sign is positive. (If Bj sign is negative, the shift will be left.)
- 43 This instruction forms a mask by RIGHT shifting a "one" from bit 59 the number of places specified by jk. Thus, RIGHT shift is always forced.

26 and 27 The Unpack and Pack instructions do not require shifting, but since all three AND gates (J, K and L) and the NORMALIZE condition will be "ones", a RIGHT shift signal is distributed to the shift network. But, these instructions do not gate the operands through the shift network and consequently, they are not shifted.

The outputs of Shift Direction Control are summarized with the following table:

DIRECTION	PIN	BOOLEAN FORMULAS
Left	9	$(\text{NORM}) + (\text{jk}) (\text{LEFT}) + (\text{NOM}) (\text{NEG}) (\text{RIGHT}) + (\text{NOM}) (\text{POS}) (\text{LEFT})$
Right	4	$(\overline{\text{NORM}}) (\overline{\text{jk}} + \overline{\text{LEFT}}) (\overline{\text{NOM}} + \overline{\text{NEG}} + \overline{\text{RIGHT}}) (\overline{\text{NOM}} + \overline{\text{POS}} + \overline{\text{LEFT}})$

Figure 7.2-11 shows how the shift direction and magnitude are combined to enable a given rank of the Shift Network. This example is for bit 2^2 or the Shift 4 places rank of the Network. The same method is used for the other ranks of the shift network.

7.2.6 SHIFT NETWORK

The shift network shifts 60-bit quantities left or right on the basis of a 6-bit shift count in the shift count (SK) register. Left shifts are circular; right shifts are end-off with sign extension.

The quantity to be shifted is transferred from the chassis input register (IR) to the shift input register (SIR) whose slave outputs drive the 100 nsec static shift network. (See Figure 7.2-12) The network is organized in six shift paths or levels of 1, 2, 4, 8, 16, and 32 shifts progressing out from the input register. Each level corresponds to a

SHIFT DIRECTION AND
MAGNITUDE CONTROL

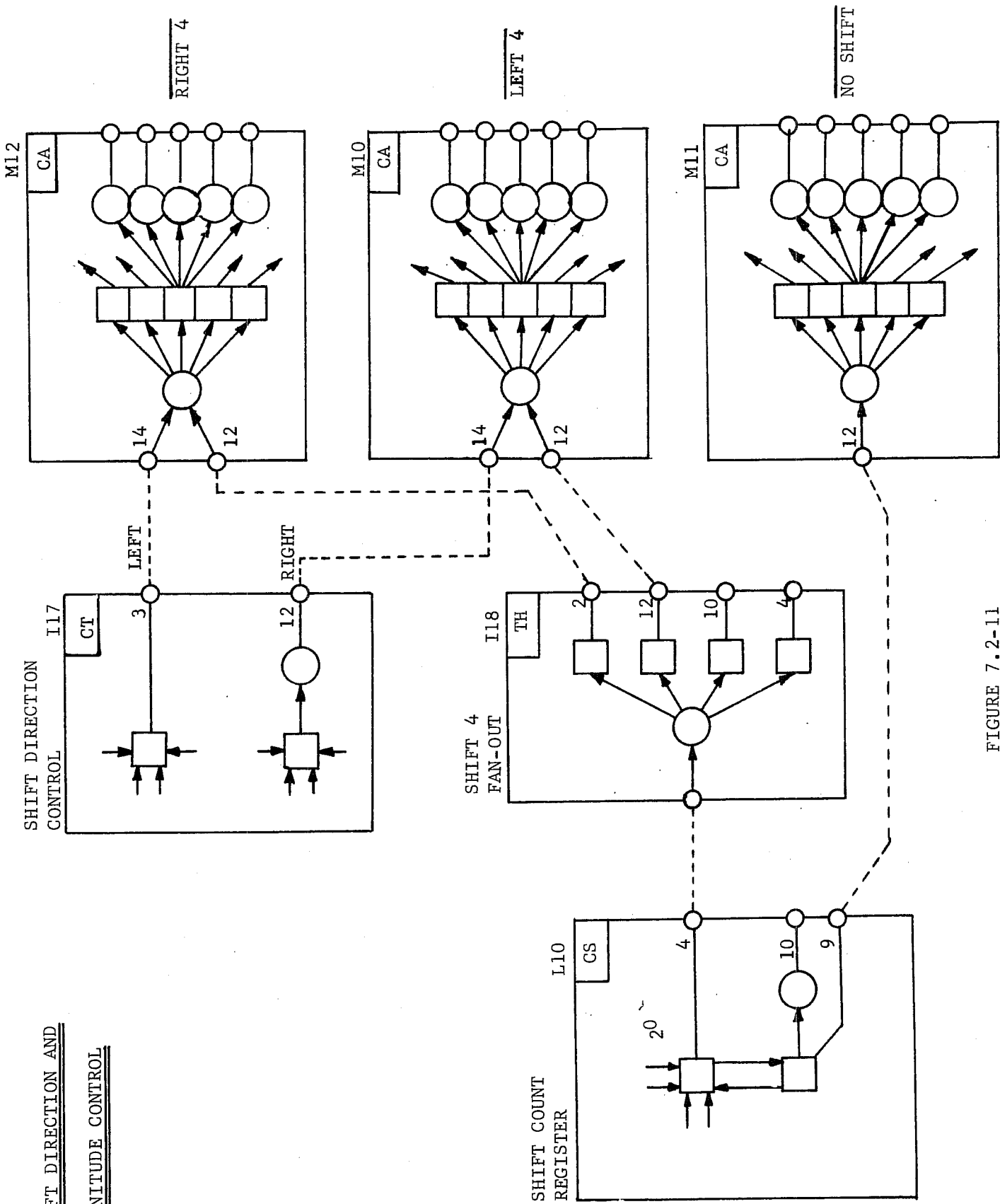


FIGURE 7.2-11

power of two and each "one" bit of the 6-bit shift count in SK gates a corresponding level of the network. A "zero" in any bit position of SK implies no shift (See Figure 7.2-11). Slave inverters on each bit of SK are gated by shift direction (left or right) according to instruction requirements. (Refer to Section 7.2.5, SHIFT DIRECTION CONTROL) The inverter outputs are fanned out to the proper level in the network and indicate a left, right or no shift for each of the six levels.

Each level of the shift network sends a bit to the next higher order shift level unshifted or shifted left or right the number of places assigned to the level. For left shifts, high order bits (2^{59}) are wired to low order positions (2^0) to provide a complete circular left shift. For right shifts, no connections are made on the right shift outputs of bit 0 or other bits in the network where a right shift would carry past bit 0.

For example, the right shift 16 output of the bit 9 circuit has no termination. This wiring produces the end-off feature of the right shift.

The sign (Bit 59) of the shifted quantity is extended on right shifts. A "one" (negative quantity) or a "zero" (positive quantity) in this position is extended to the right the number of positions the quantity is shifted. During the formation of a Mask, a negative sign is forced. The shift network treats zero extension automatically. Since any shift will be at least one position, ones are extended beginning at the second level and the first level is ignored. The sign extension slaves are thus necessary only when the shift quantity is negative.

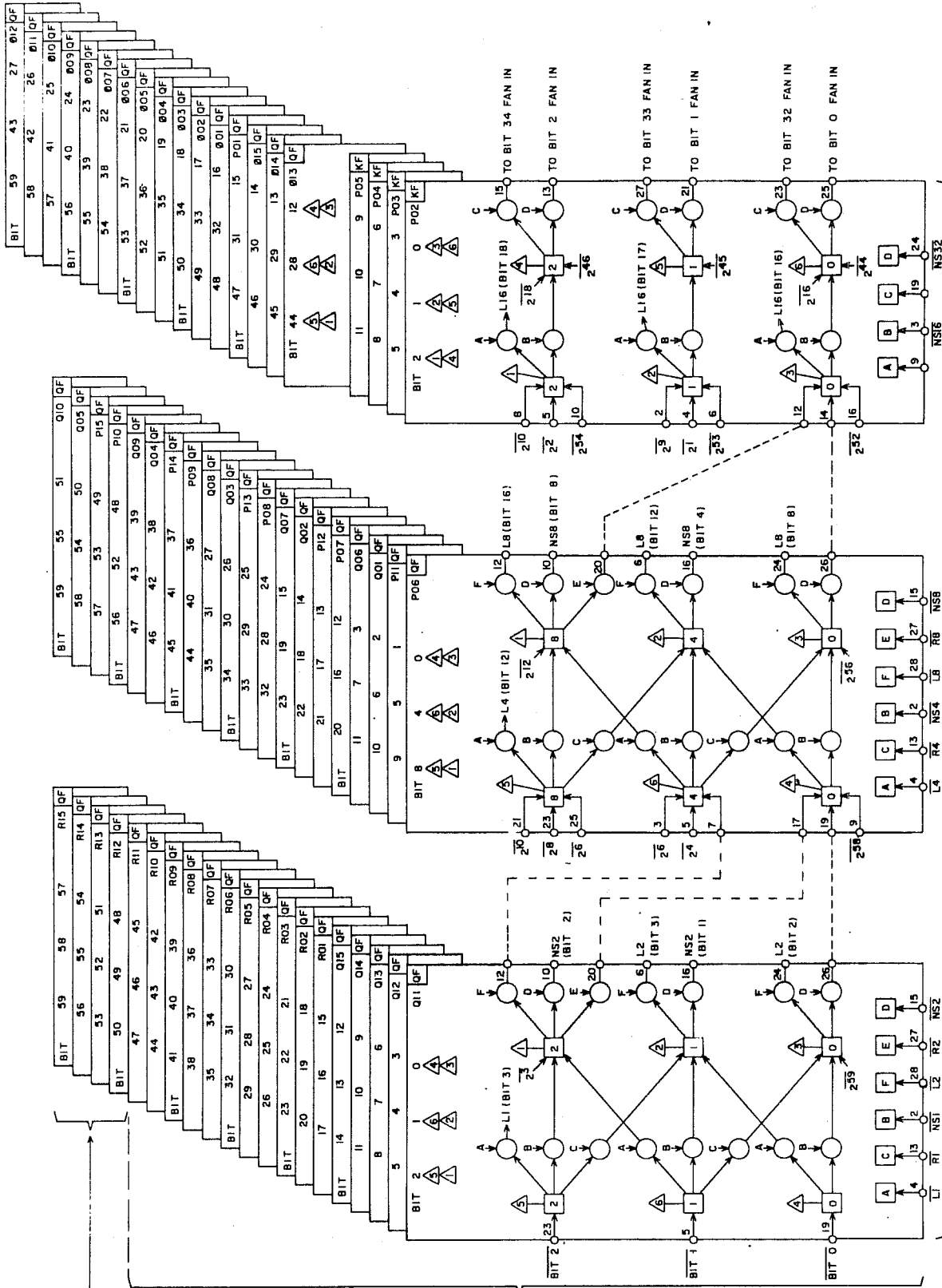
Figure 7.2-12 is a representative logic drawing of the Shift Network showing bits 0, 1 and 2. Notice the three enables (Left, Right, or No Shift) entering the first four stages of the Network. Note also, that on stages 16 and 32 only two enables enter each stage, Left and No Shift, since a right shift of 16 or 32 would produce an end off effect for these stages. The design of the network is like that of the Peripheral Processor Shift Network with the exception that this is a 60-bit, 6 stage network. The PPU shifter was an 18 bit, 5 stage network.

CHASSIS & SHIFT
INPUT REGISTERS

EXPONENT
12 BITS

COEFFICIENT
48 BITS

$$x_i + x_{i-1}$$



NOTE:
L = LEFT
R = RIGHT
NS = NO SHIFT

Shift Network

Figure 7.2-12

7.2.7 NORMALIZE NETWORK

The static normalize network forms a six-bit shift count, which defines the number of shifts necessary to normalize, and stores the count in the SK register. The shift network, under control of the SK register, is used to normalize the coefficient.

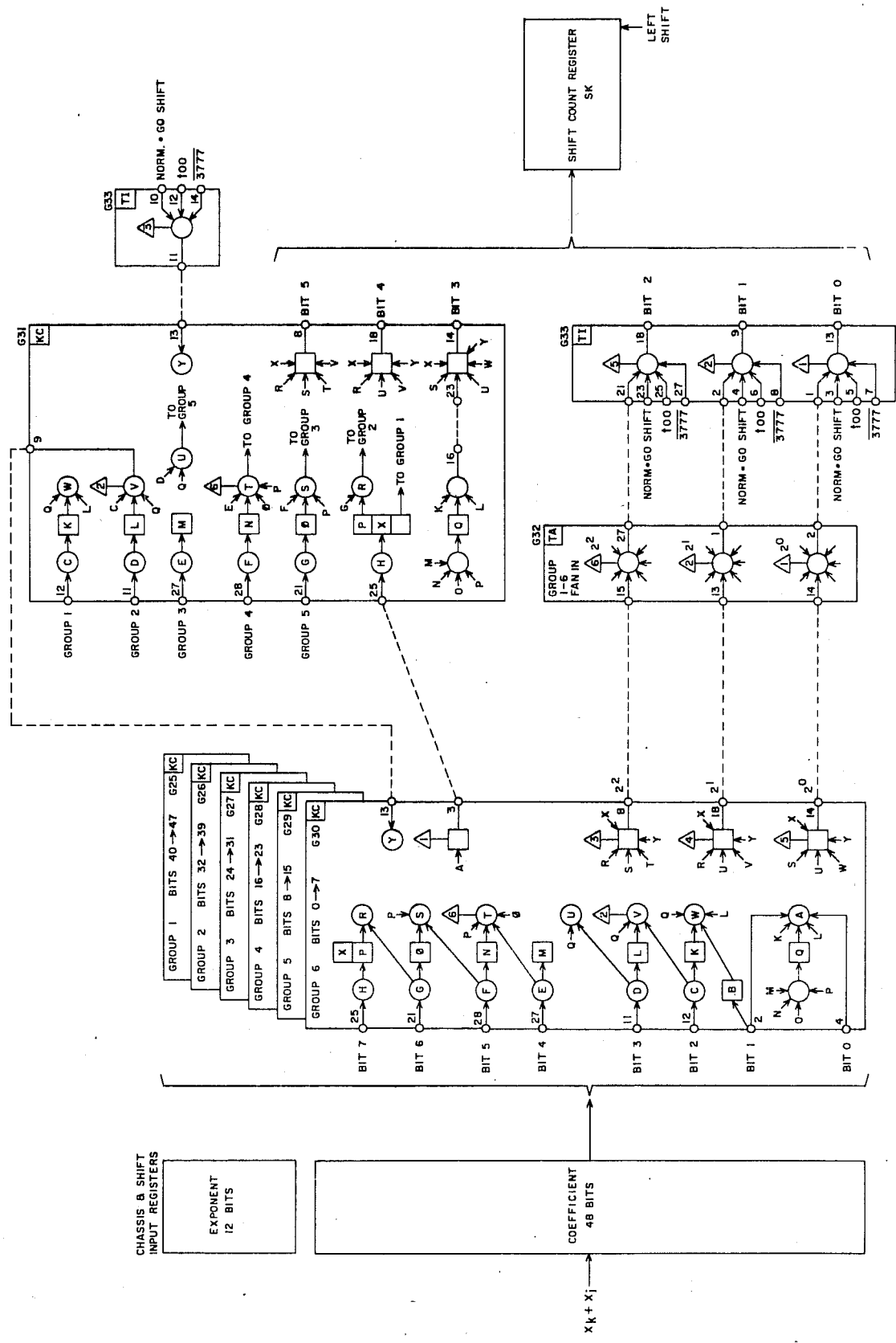
Initially, the coefficient is transferred from the chassis input register to the SIR (complemented if X_k is negative). Thus, the content of SIR is always positive during normalize operations. The normalize network organizes the low-order 48 bits of SIR into six 8-bit groups. It then:

1. Determines the highest order "one" in each group (a 1 of 8 selection).
2. Determines the highest order group with a "one" (a 1 of 6 selection).
3. Determines the number of zeros between the highest order "one" of the coefficient and bit 2^{47} , and stores the quantity in the SK register.

Refer to Figure 7.2-13, during the following discussion.

Locating the highest order "one" in a group is a 1 of 8 selection which is accomplished by comparing a bit with all of the higher order bits in the group (modules G25-G30). In each of the six groups, this yields a three-bit "group count" which indicates the number of shifts necessary to move the bit to the most significant position in its group (test points 3, 4, and 5). The six group counts are combined in an OR circuit (TA module) which feeds the lower three bits of the SK register. The group count selected to enter SK corresponds to the group holding the highest order "one" in the coefficient.

Another circuit in each group tests the group for "All zeros" (modules



Normalize Network
Figure 7.2-13

G25-G30, inverters A). The highest order group with a "one" is determined in the same manner as described above, i.e. each group is compared with all higher order groups through the all zeros circuits (module G31). Again, a three-bit quantity is formed in a count network but is sent to the upper half of the SK register. This quantity can be thought of as representing the number of sequential all zero groups beginning with the most significant (group 1).

Summary: Starting with the high-order bits of the coefficient, groups with all zeros are eliminated. When the first group containing a "one" is found, a count is sent to the three low-order bits of SK. This count is equal to the number of places required to shift the "one" to the highest-order bit of the group. The upper three bits of the SK register are loaded with the number of all zero groups to the left of the group containing the first one. Entry of all other groups into SK is blocked. Thus, a six-bit shift count ranging from 0 to 48 (60_8) is formed in SK.

7.2.8 B_j ONES TEST NETWORK

The purpose of this test is to guarantee an all zero coefficient when the shift count contained in B register j exceeds 77_8 and a Nominal Right Shift is specified. For example, assume that B_j equals 000425_8 and a 23 (Shift Right Nominally) instruction is coded. The proper (all zero) coefficient should be obtained by shifting the coefficient 425_8 places to the right (since right shifts are by nature end-off). Since only six bits are contained in the Shift Count Register, the maximum shift possible is 77_8 places. In the case of a Nominal Shift, only the lower six bits of B register j are used, and thus if the Ones Test Network was not present, a Shift count of 25_8 would be used in this example. The result could obviously be erroneous since only 21_{10} bits of the co-

efficient would be shifted to the right and end-off. The possibility of such an error occurring is eliminated by the Bj Ones Test Network.

This network is used to determine whether any one of Bj bits 6 through 10 is set during Nominal Right Shifts. The question arises: why are only bits 6 through 10 tested? The answer is found by considering that the Left Nominal (22) instruction is used during the conversion of Floating Point Numbers to Integers. The following example illustrates this process:

GIVEN: (X3) = 2006 0 \longrightarrow 0527

PROGRAM STEPS: 26423 (Unpack X3 to X4 and B2)
22624 (Left Shift X4 Nominally B2 places to X6)

- RESULT:
1. The Unpack instruction will place 527 in X4 and 6 in B2.
 2. The Shift Nominal instruction will shift X4 six places to the left and place the number 52700 in X6.
 3. Thus, the floating point number 527×2^6 is converted to the Integer 52700.

The point to be stressed from this example is that the Unpack instruction will not return to Bj, an exponent greater than 10 bits in magnitude. Consequently, the Shift Nominal instruction, if used properly, should never find an exponent (Bj Register) greater in magnitude than 10 bits.

Figure 7.2.14 is a logic drawing of the Bj Ones Test Network. The test is a relatively simple matter of checking the state of bits 6 through 10 together with the sign (2^{17}) of B register j. A "one" out of either test point 5 or 6 will disable the sending of the shifted coefficient to the output network since a "one" is required on pin 10 of H24. Thus, to enable the shifted result, the flip-flop on K30 must NOT be set. Test point 5 checks the Negative ("one") state of Bj against a

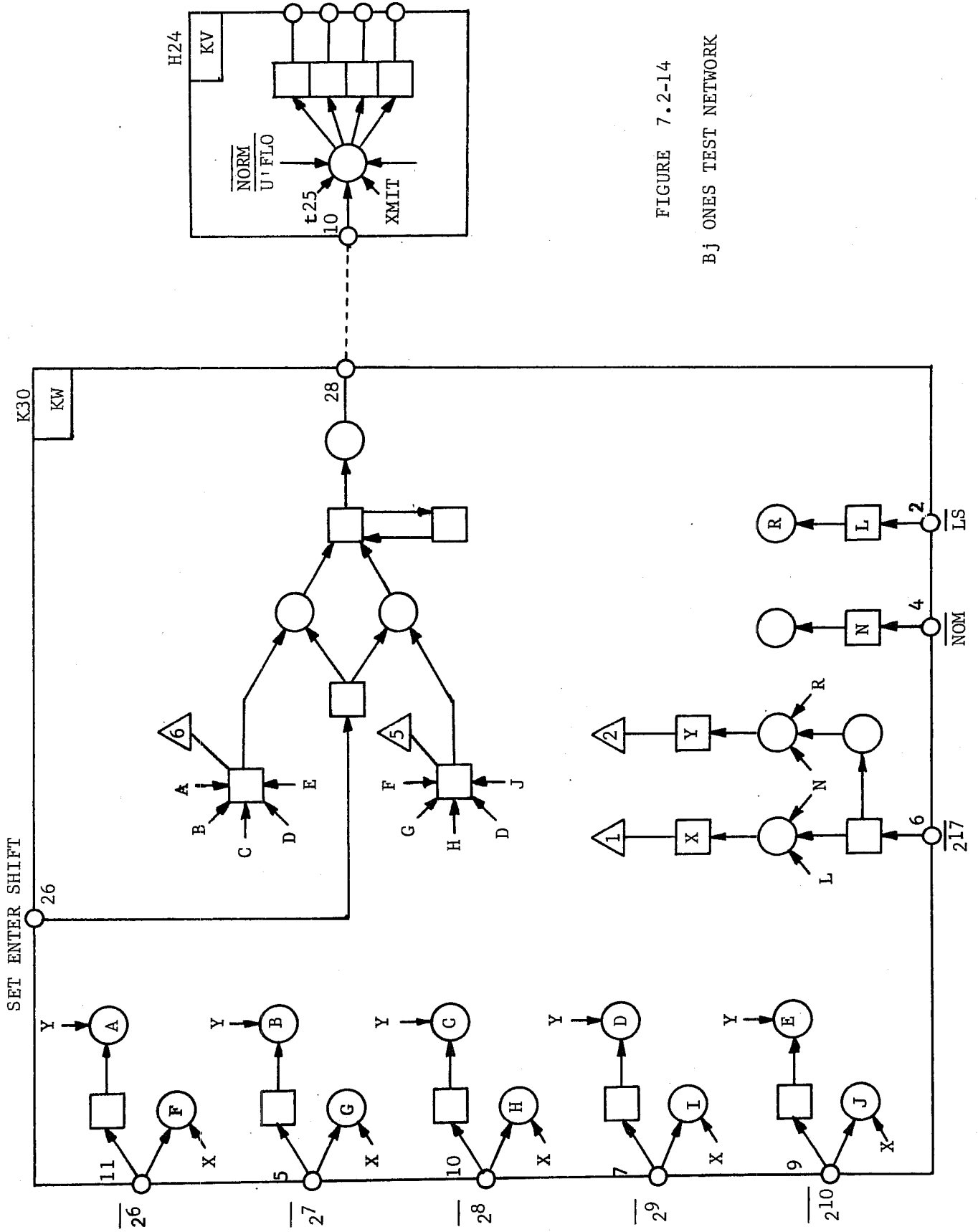


FIGURE 7.2-14

BJ ONES TEST NETWORK

zero in each of bits 6 through 10 (terms "X" and "F", "G", "H", "I", and "J"). If B_j is negative, zeros are checked because the magnitude is in complement form. If any of these bits is a zero when B_j is negative, a zero is forced into test point 5 and when the "Set Enter Shift" is enabled, the flip-flop on K30 will be set. Test point 6 checks the Positive ("zero") state of B_j against a "one" in each of bits 6 through 10 (terms "Y" and "A", "B", "C", "D", and "E"). If any one of the bits is a "one" when B_j is Positive, a zero into test point 6 will allow the flip-flop to set when the "Set Enter Shift" gate arrives. Note that this circuit is used only during Nominal shifts when the Right direction is specified. The following Boolean formulas express the conditions required for terms "X" or "Y".

$$X = (\text{NOMINAL}) (\text{LEFT}) (2^{17}) \text{ or, Nominal Left and } B_j = \text{Neg (Right Shift)}$$

$$Y = (\text{NOMINAL}) (\overline{\text{LEFT}}) (\overline{2^{17}}) \text{ or, Nominal Right and } B_j = \text{Pos (Right Shift)}$$

Thus, during Nominal Right Shifts when there is a one present in any of bits 6 through 10 (true magnitude) the shifted result is not enabled to the coefficient bits of the transmitters. An all zero coefficient is then returned to X_i .

7.2.9 EXPONENT ADDER

Since the coefficient of X_k is shifted left (increased) during the normalize process, the exponent of X_k must be decremented by an equivalent value. This is the function of the exponent adder - to subtract the shift count generated by the normalize network from the exponent of the source operand. The example in figure 7.2-15 illustrates the initial and final values of a normalize operation. In the example, a normalize

shift count of 37_8 is generated and sent to the SK register, which in turn conditions the shift network. The six bits of SK also feed the exponent adder along with the eleven bits ($2^{48} - 2^{58}$) of the original exponent.

Before Normalizing:

$$(X_k) = 2135 \text{ 0-----}0327715 \quad (327715 \times 2^{135})$$

After Normalizing:

$$(X_i) = 2076 \text{ 6576320-----}0 \quad (657632 \times 2^{76})$$

$$(B_j) = 0\text{-----}037 \quad (\text{Normalize Shift Count})$$

Figure 7.2-15

Recall, that if the sign of the coefficient is negative during normalize operations, the complement of IR is sent to SIR. If the sign is positive, the true value of IR is sent to SIR. Thus, bits 48 - 58 of SIR will always contains the true form of the biased exponent and bit 59 will always be zero. The sign of the exponent can be determined with bit 58; the set condition indicated positive with the magnitude in true form (2000 - 3777) and the cleared condition indicates negative with the magnitude in complement form (0000 - 1777).

Also recall that underflow may occur during the normalize process, but only if the exponent is smaller (more negative) than negative 60_8 (since 60_8 is the largest shift count that can be generated by the normalize network).

For the purpose of explanation, the range of possible exponents is divided into four groups:

- 1) 0000 - 0057 (-1777 to -1720): A Negative exponent, and

under flow can occur if $SK > {}_5EXP_0$. It is indicated by the presence of an End Around Borrow (EAB).

UNDERFLOW	UNDERFLOW
EXP = 0032 -1745	0053 -1724
SK = <u>51</u> - 51	<u>15</u> - 15
RESULT = 1161 -2016	0036 -1741
EAB? YES	NO

2) 0060 - 1777 (-1717 to -0000)

Negative, but no underflow can occur (an End Around Borrow is not possible).

EXP.	0060	-1717
MAXIMUM (SK)	<u>60</u> 0000	- 60 -1777

3) 2000 - 2057 (+0000 to +0057)

Positive, and transition to negative might occur (if $SK > {}_5EXP_0$)

	<u>Transition</u>	<u>No Transition</u>
EXP	2050 +50	2046 +46
SK	<u>52</u> -52 1776 (forced) -02	<u>35</u> -35 2011 +11
	<u>1</u> 1775	

4) 2060 - 3776 (+0060 to +1776)

Positive, but transition to negative is not possible.

EXP	2060	+60
(SK) Maximum	<u>60</u>	-60
	2000	+00

With this information in mind, a discussion of the adder logic follows. The terms "Generate", "Satisfy", and "Enable" are used during this explanation. They refer to the 1 and 0 combinations in each stage and are defined as follows:

Generate: requires (generates) a borrow from a higher stage.

Satisfy: will fill (satisfy) the borrow requirement of a lower stage and in doing so, will not generate a borrow.

Enable: in a sense, full-fills a borrow requirement, but in so doing, generates a borrow itself. It therefore passes-on (enables) a borrow to the next higher stage.

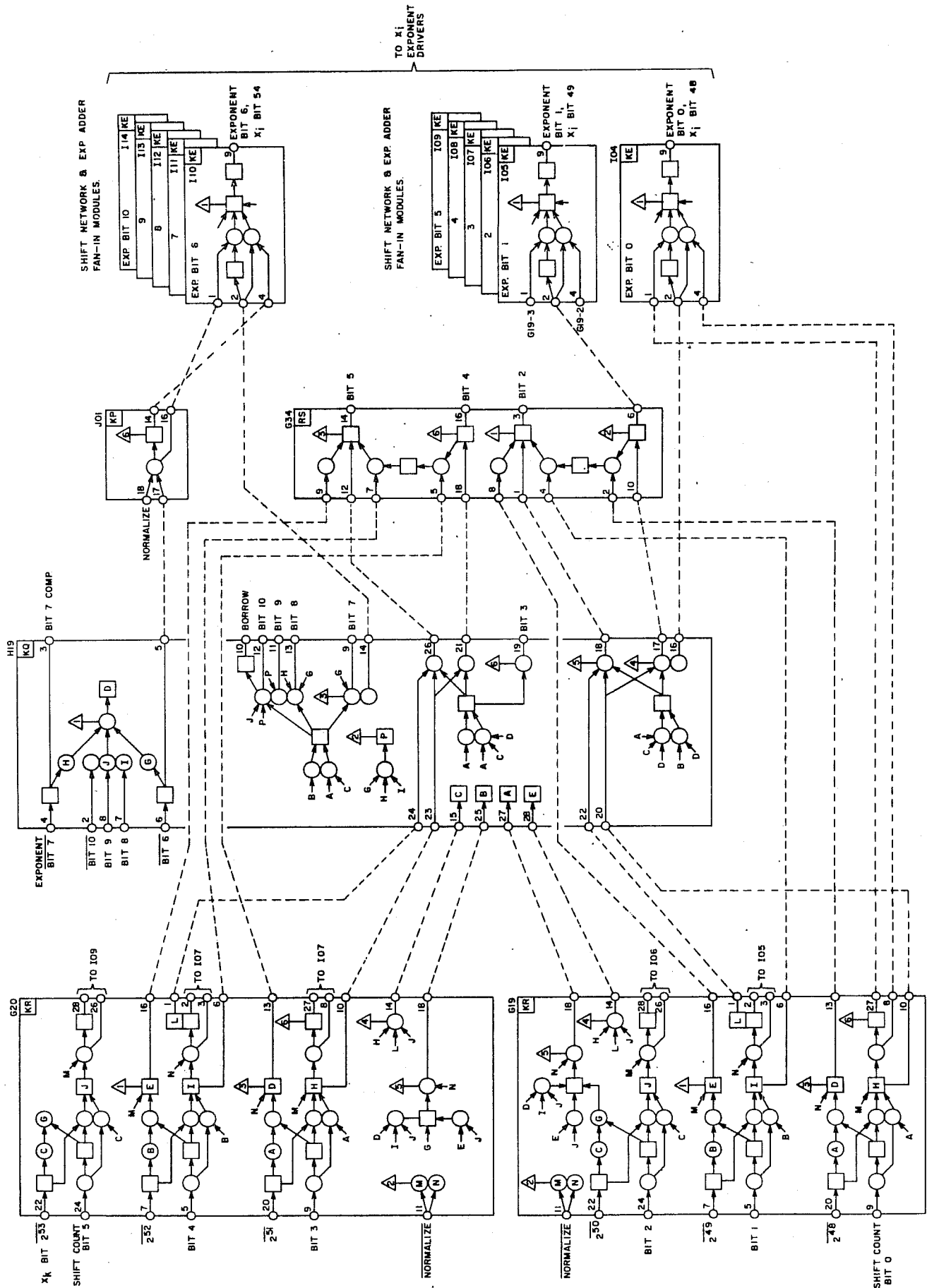
Figure 7.2-16 shows the subtraction of SK from the Xk exponent of our original example (FIGURE 7.2-15). Each stage is labeled as to its state; i.e. Generate, Satisfy, or Enable. Note that bits 6-10

Xk exp. =	10 9	876	543	210
	S E	EES	EEE	EGE
	1 0	001	011	101
SK =				
	1 0	000	111	110

FIGURE 7.2-16

of SK are not physically present and are therefore considered to be zeros. Consequently, a generate in stages six through ten is an impossibility. The zero/zero enable and satisfy are the only conditions possible in these stages. Thus, if a borrow is required by stage 5, it can be satisfied by a "one" in any of bits 6 - 9 of the Xk exponent. If no satisfy exists, all enables are assumed, each of bits 6-9 is toggled to a one, bit 10 is cleared, and an End Around Borrow is generated. Study the following example.:

Xk =	2031	=	10	000	011	001	=	+31	
Sk =	55	EAB =			101	101	EAB =	-55	
	1754	=	01	111	101	100	=		
	1	=	01	111	101	011	=	-24	
	1753								



Exponent Adder
Figure 7.2-17

Recall, that an EAB may also be generated if an exponent lies in the range, 0000 - 0057 (Group 1), but in this case, it indicates the Underflow condition. Thus, an EAB may occur in two cases, 1) with a positive exponent in the range 0-57 and 2) with a negative exponent in the range - 1777 - 1720. The adder logic differentiates between the two with the following conditions expressed in Boolean:

1. (EXP bit $2^{10} = 1$) (EAB) \rightarrow Positive to Negative transition.
2. (EXP bit $2^{10} \neq 0$) (EAB) \rightarrow Underflow.

Figure 7.2-17 is a logic drawing of the exponent adder. To the left of the drawing, two KR modules (G19 and G20) determine the generate, satisfy, and enable condition for each of the low-order six stages. The outputs of inverters H, I, or J being a "one" indicate the enable (equivalence) condition for stages 0, 1, and 2 (G19) and 3, 4, and 5 (G20). Translations for the generate condition are also made on these modules (pins 13, 16, and 18 on both G19 and G20 indicate the generate condition).

The KQ module (H19) determines the effect of an EAB on each stage of the adder. It determines whether or not an EAB is generated and checks for satisfies in each of the bit positions. A "one" out of term "D" indicates a 10 000 (20XX) configuration in bits 10 - 6 of the exponent and will enable the propagation of an EAB through bits 0-9 of the adder. A "zero" out of term "D" indicates that the exponent is negative of a satisfy is present in bits 9-6 ($\bar{D} = \bar{10} + 9 + 8 + 7 + 6 = \overline{20XX}$) and disables the propagation of an EAB (Recall that if an EAB occurs when the exponent is negative, underflow has occurred) Pin 10 of H19 is the EAB signal that is ANDed with the $\overline{2^{10}}$ condition to indicate an underflow condition (Refer to the C.E. Diagrams, sheet 111, I24, pin 6)

The RS module (G34) is a further summation network that determines whether or not a generate enters stages 1, 2, 4, or 5. This condition is determined for stage zero at pin 16 of H19, and for stages 6, 7, 8, and 9 by pins 14, 9, 13, and 11 respectively, of H19.

The KE modules (I04 through I14) are used to AND the EQUIVALENCE or $\overline{\text{EQUIVALENCE}}$ conditions with the Borrow or $\overline{\text{Borrow}}$ condition for each stage. Here, the final result for each stage is determined according to the following table:

CONDITION ON KEs	RESULT
(EQUIVALENCE) (BORROW) \Rightarrow	1
(EQUIVALENCE) ($\overline{\text{BORROW}}$) \Rightarrow	0
($\overline{\text{EQUIVALENCE}}$) (BORROW) \Rightarrow	0
($\overline{\text{EQUIVALENCE}}$) ($\overline{\text{BORROW}}$) \Rightarrow	1

SHIFT FUNCTIONAL UNIT - FLOW CHART

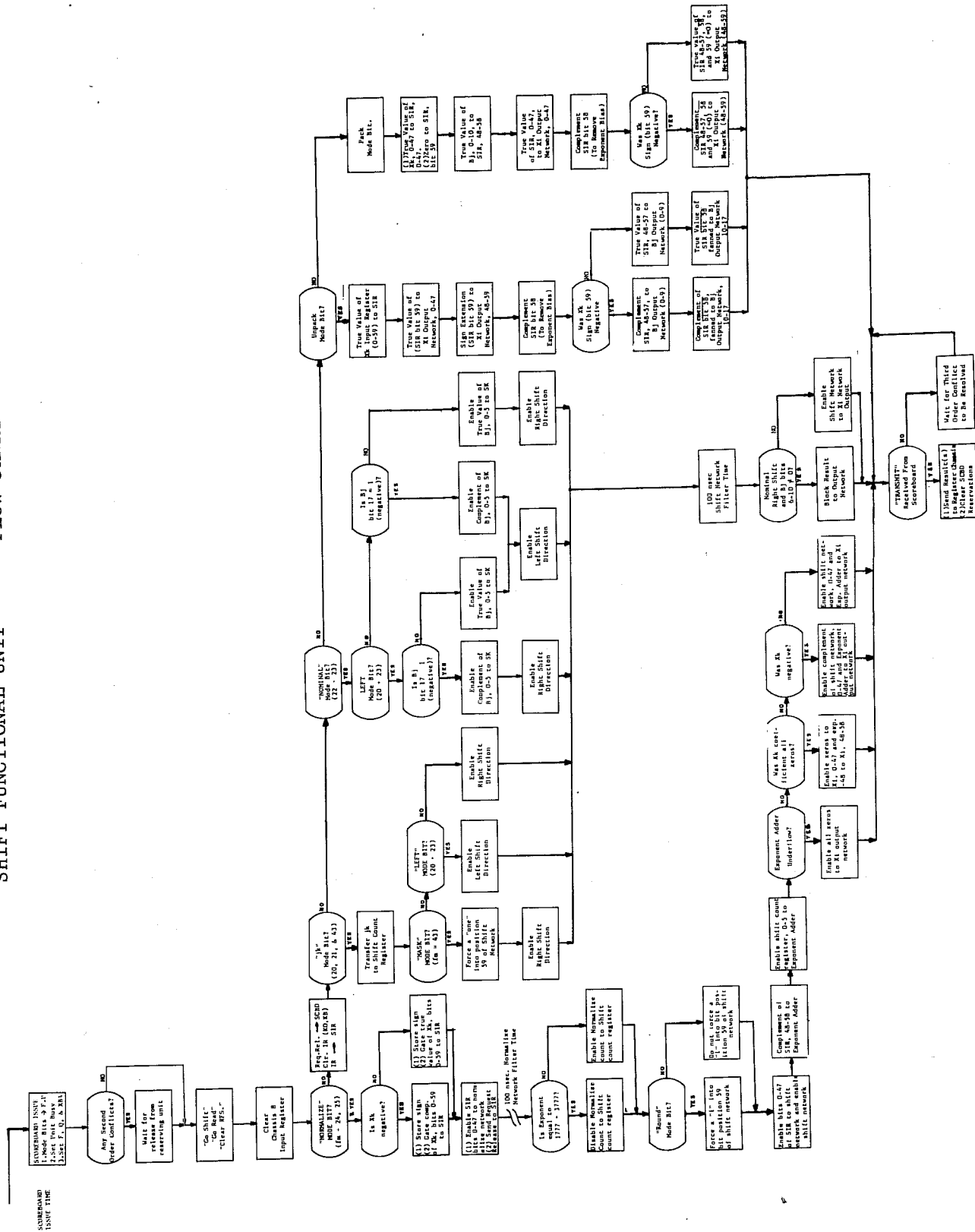


Figure 7.2-18

SECTION 7.3

LONG ADD

Functional Unit

0
1
2
3
4
5
6
7
8
9
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
[
\
]
^
_
`
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x
y
z
{
|
}
~
?
@

\$
%
&
*
(
)
+
,
-
.
:/
:;
=>
?@

\$%
&
*
(
)
+
,
-
.
:/
:;
=>

LONG ADD FUNCTIONAL UNIT

7.3.1 INTRODUCTION

The Long Add Unit is an integer arithmetic unit that performs fixed point addition and subtraction of 60-bit operands and performs tests on X registers which are used to condition the O3X jump instructions. It is a 300 nanosecond unit located on data trunk #1 along with the Floating Add and Shift functional units. Long Add holds third (last) priority for reading operands and for storing results.

The Long Add Unit is controlled by mode bits (only one), a timing chain, and the scoreboard (to the extent of starting the unit and transmitting results). It also contains an adder capable of forming a 60-bit sum or difference and testing networks which check the sign, zero, infinite, and indefinite conditions of X registers. The resulting control bits of these testing networks are sent to the Branch Unit where they enable or disable conditional jumps. The testing networks are not used during the addition and subtraction processes; no arithmetic error conditions are checked and therefore overflow, underflow, and indefinite results are ignored.

In discussing the O3X Branch instructions, it is helpful to review the events that take place during the movement of any OX instruction to the scoreboard. Recall, that in transferring a OX instruction from U1 to U2, the i and j portions of U1 are shifted to the j and k portions of U2. The i portion of U1 is also sent to the i portion of U2. Symbolically, then, the transfer looks as follows:

U1 = f m i j k k-----k

U2 = f m i i j k-----k

Translations used to determine the existence of a result register (first order) conflict are made from the U2 i portion and Result (Ai, Bi, Bj or Xi) flip/flops. In the case of OX instructions none of the four Result flip/flops are set, hence the issuance of a OX instruction cannot be delayed by a Result register conflict. (The functional unit type of first order conflict may exist, since in the case of the O3X instructions the Long Add unit must be used, but is not significant to this discussion).

In the discussion of second order (source operand) conflicts we will analyze, specifically, the case of the O3X instructions.

Since the Long Add unit must read the X register designated by the k portion of U2 (the j portion of the original instruction format) the scoreboard must determine whether or not that register is reserved for the result of another functional unit. This is done in the standard manner -- by transferring the XBA designator for the specified X register to the QK designator of the Long Add unit and translating Q. If $Q = 0$, the register is not reserved and RF2 (XK) will be set. If $Q \neq 0$, the register is reserved by the unit whose code is in QK, and the read flag will be set when that unit is released. Although only one operand is read during O3X instructions, both read flags must be set to generate the "Go Read" signal for the long add unit. This means that Read Flag 1 (Xj) must also be set even though the Xj operand will not be used. Since no special gates exist for setting RF1 for this case, the normal procedure of setting and translating Qj is used. Consequently, a second order conflict may occur if the X register specified by the i portion of the original instruction is reserved.

Normally, when these second order conflicts are resolved, two "Go Read" signals (X_j and X_k) and two 3-bit tags (F_j and F_k) are sent to register Exit control to gate the source operands to the Long Add Unit. Upon issuing 03X instructions to the scoreboard, only the F_k designator is set (setting F_j and F_i is disabled by the translation, $f_m = 0X$). Also, setting RF_1 and RF_2 results in sending only the "Go Read" and F designator for X_k to Exit control. (The F_j designator, which equals zero, is sent to exit control, but the absence of the "Go Read X_j " inhibits translation; therefore all zeros are sent on the Long Add X_j data trunk). In conclusion, both the j and k octals of U_2 may specify an X register which is reserved, and thus cause a second order conflict. Nevertheless, once the conflicts are resolved only a "Go Read X_k " is sent to Exit Control and the Long Add Unit receives only the one operand originally designated by the j portion of the 03X instruction.

To consider the possibility of a third order conflict arising during 03X instructions; recall that due to the $f_m = 0X$ translation, the F_i designator of the Long Add Unit, is not set at scoreboard issue time. F_i will thus contain "0". Also, because of the 0X instruction, translation of $F_i = 0$ is inhibited (by the clear side "Full Bit" for F_i). When the "Request Release" arrives at the All Clear Network there is no Long Add F_i translation to compare with other F_j and F_k designators. Thus, a third order conflict cannot be generated, and the Long Add unit will immediately be sent the "Transmit" signal. As a result, the 60-bit quantity in the Long Add output network is gated to register entry control. Because the Full Bit of the F_i designator is cleared, a "Go Store" to Entry control is inhibited, although the three bit F_i designator (equal to zero) is sent. In order

to translate a code $F_i = 0$, the translating network in Entry control requires a "Go Store" pulse. Since one was not sent, the translation is disabled and none of the gates to the X registers are opened. To summarize, during 03X instructions a third order conflict cannot be generated. Although a "Transmit" is sent to Long Add, no Entry tag is translated and the quantity transmitted is therefore lost in Entry control.

Since the Long Add F_j and F_i "Full Bits" remain cleared only during the 03X instructions, first, second, and third order conflicts for the 36 and 37 instructions are handled in the conventional manner.

LONG ADD UNIT - BLOCK DIAGRAM

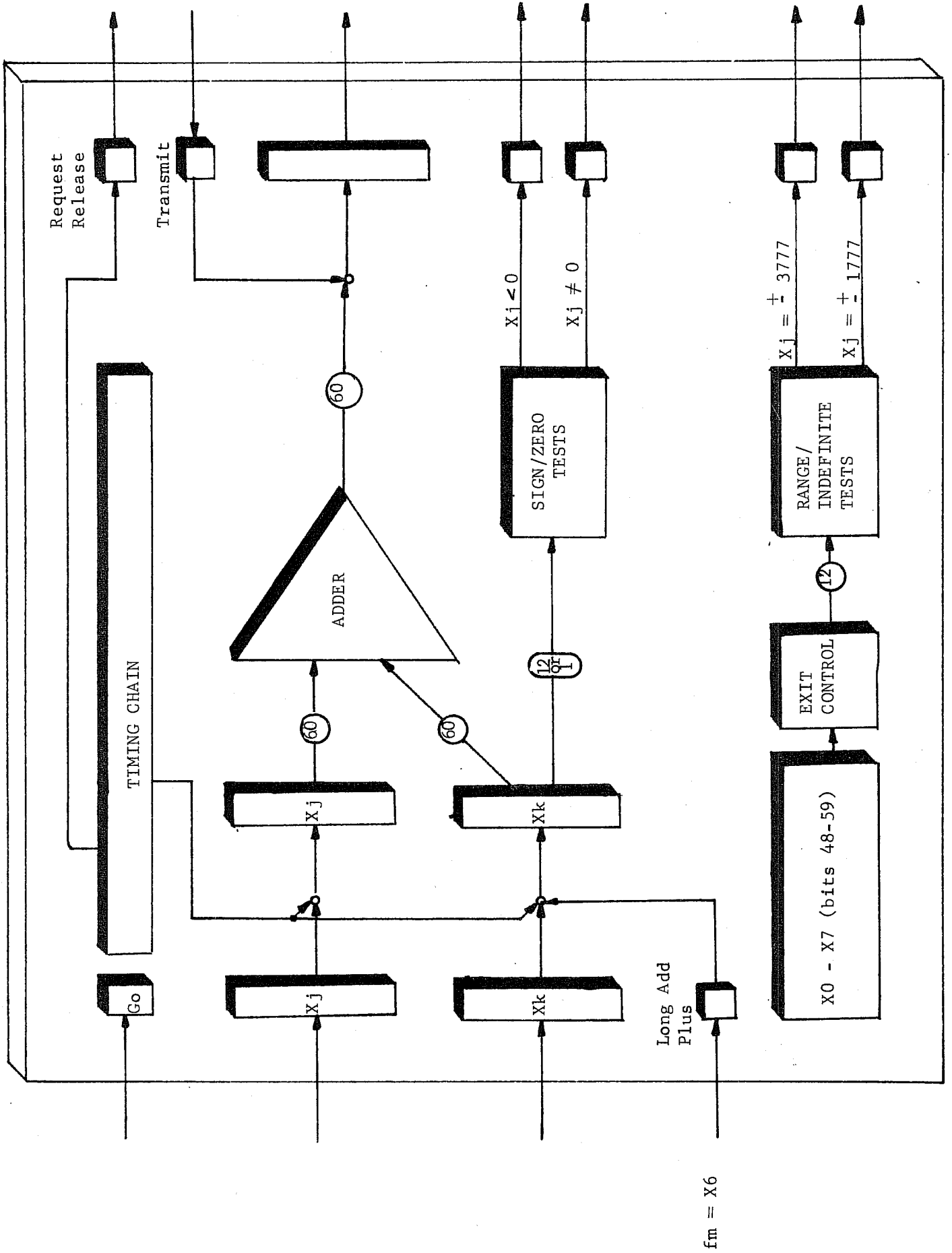


Figure 7.3-1

7.3.2 INSTRUCTION LIST/DATA FLOW

This discussion is divided into two subsections, (1) Fixed Point Arithmetic Instructions and (2) Conditional Branch Instructions. The expressions in parenthesis which follow the instruction name are the symbolic ASCENT Assembler codes. Data flow may be followed by referring to the block diagram, Figure 7.3-1.

Fixed Point Arithmetic Instructions

36 Integer Sum of X_j and X_k to X_i ($IX_i = X_j + X_k$)

Definition:

Forms a 60-bit one's complement sum of the quantities from operand registers X_j and X_k and stores the result in X_i . An overflow condition is ignored.

Data Flow:

The source operands, X_j and X_k are transferred from the chassis 8 input register to the feeder registers of the long add functional unit.

Both X_j and X_k are transferred in true form. When a "transmit" signal is received from the scoreboard, the sum of X_j and X_k is gated to the result register X_i .

37 Integer Difference of X_j and X_k to X_i ($IX_i = X_j - X_k$)

Definition:

Forms the 60-bit one's complement difference of the quantities from operand registers X_j (minuend) and X_k (subtrahend) and stores the result in X_i . An overflow condition is ignored.

Data Flow:

The source operands, X_j and X_k are transferred from the chassis 8

input register to the feeder registers of the Long Add functional unit. X_j is transferred in true form; X_k in complement form. When the "transmit" signal is received from the scoreboard, the difference of X_j and X_k is gated to the result register.

Conditional Branch Instructions

030	JUMP to K if $X_j = 0$	(ZR X_j K)
031	JUMP to K if $X_j \neq 0$	(NZ X_j K)
032	JUMP to K if $X_j =$ plus (positive)	(PL X_j K)
033	JUMP to K if $X_j =$ negative	(NG X_j K)
034	JUMP to K if X_j is in range	(IR X_j K)
035	JUMP to K if X_j is out of range	(OR X_j K)
036	JUMP to K if X_j is definite	(DF X_j K)
037	JUMP to K if X_j is indefinite	(ID X_j K)

Definitions:

These instructions jump to address K when the 60-bit word in operand register X_j meets the condition specified by the i digit.

Test Validity:

030 & 031 - The zero test check the full 60-bit word in X_j . The words 0---0 and 7---7 are considered as zero. All other words are non-zero. The test is therefore valid for both fixed and floating point words.

032 & 033 - The sign tests check only bit 2^{59} (sign) of X_j . A zero indicates positive; a one indicates negative. The test is valid for both fixed and floating point quantities.

034 & 035 - The range tests check the upper 12 bits ($2^{59} - 2^{48}$) of X_j for both plus (3777X----X) and minus 4000X----X) infinity. Since the low order 48-bits are ignored, near

overflow numbers are also considered out of range. The test is valid for both fixed and floating point numbers.

036 & 037 - The definite/indefinite tests check the upper 12 bits ($2^{59} - 2^{48}$) of X_j for both plus (1777 X-----X) and minus (6000 X----X) indefinite forms. The test is valid only for floating point values.

Data Flow:

During the 03X instructions, the operand to be tested is sent to the X_k feeder register in complement form. There is input to the X_j input register; it therefore contains all zeros. For sign tests, (32 + 33) a test of X_k bit 59 will indicate a negative or positive quantity. For the zero tests (030 + 031) all 60-bits of X_k are checked for all "zeros" or all "ones". Either condition indicates a "zero" quantity.

During Range or Indefinite tests, the Input Registers are not used. Instead, the upper 12 bits of X_k are checked from the operating register (through Exit control). The Range tests checks bits 48-59 for 3777 or 4000 (positive or negative infinity). The indefinite tests checks these bits for 1777 or 6000 (positive or negative indefinite). The presence or absence of control signals resulting from these tests are where they are sent to the Branch Unit to enable or disable the 03X conditional jump instructions.

7.3.3 MODE BIT

As far as arithmetic processes are concerned, the Long Add unit is capable of performing only two; addition and subtraction. A single mode bit, called Long Add Plus is used to distinguish between these operations. It is generated with $fm \text{ translation} = X6 \text{ ANDed with Long Add Unit Busy}$. The true value of operand X_j is always sent to the X_j feeder register. During Addition, the Long Add Plus flip/flop is set and enables the true value of operand X_k to the X_k feeder register. Since the adder is additive in nature (it forms $X_j \text{ feeder plus } X_k \text{ feeder}$) the sum of X_j and X_k will be generated. During Subtraction, the Long Add Plus flip/flop is cleared. Consequently, the complemented value of X_k is sent to the X_k feeder and the difference, $X_j - X_k$, is formed.

During the Branch (03X) instructions the Long Add Plus flip/flop is cleared. Therefore, the complemented value of the the X_k input register is sent to the X_k feeder, whose outputs are used in making the sign and zero tests. Range and Indefinite tests are made directly from the operating register (through Exit Control) j hence the mode bit is not significant.

Figure 7.3-2 is a logic drawing which shows the effect of the Long Add Plus mode bit on the IR to feeder transfers.

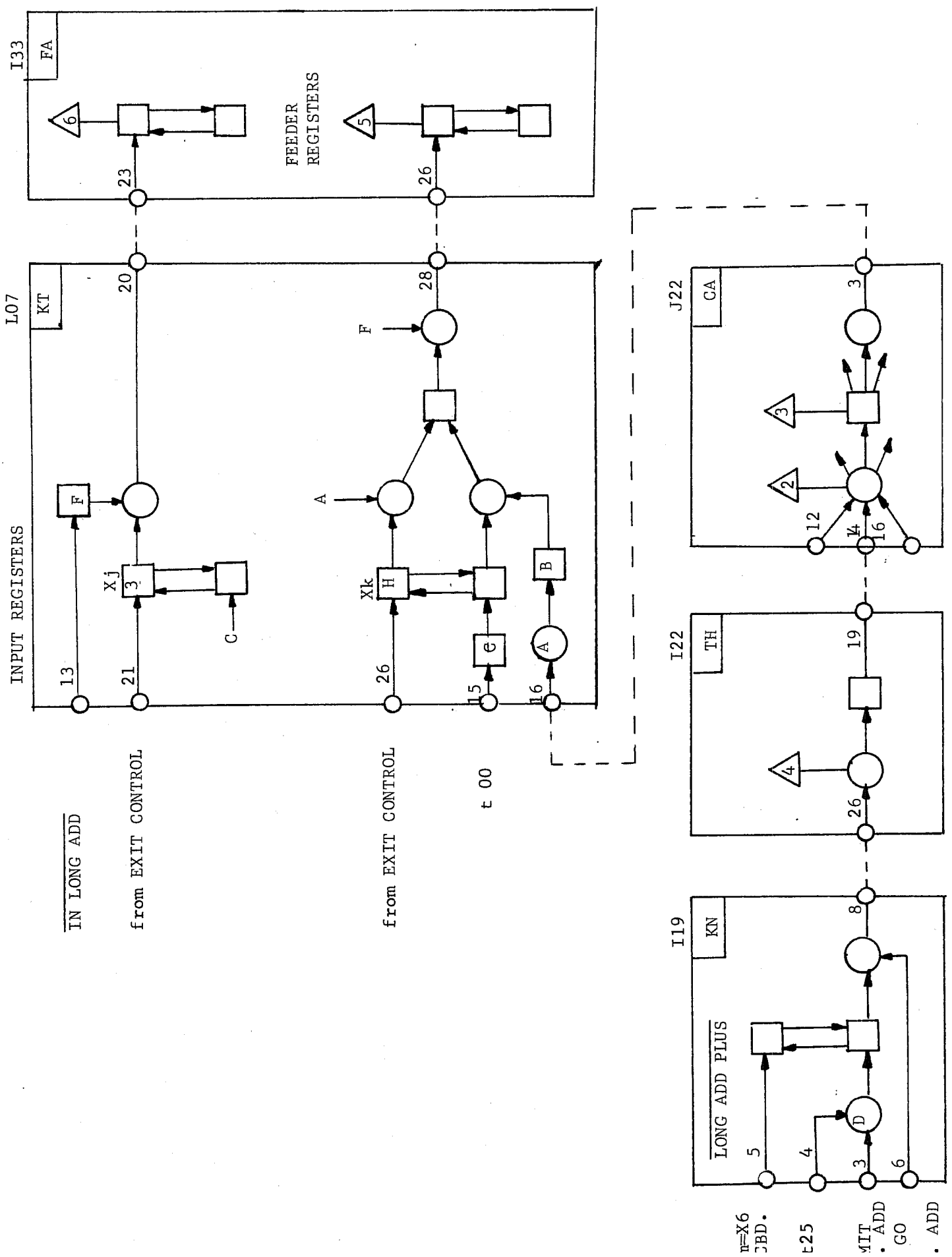


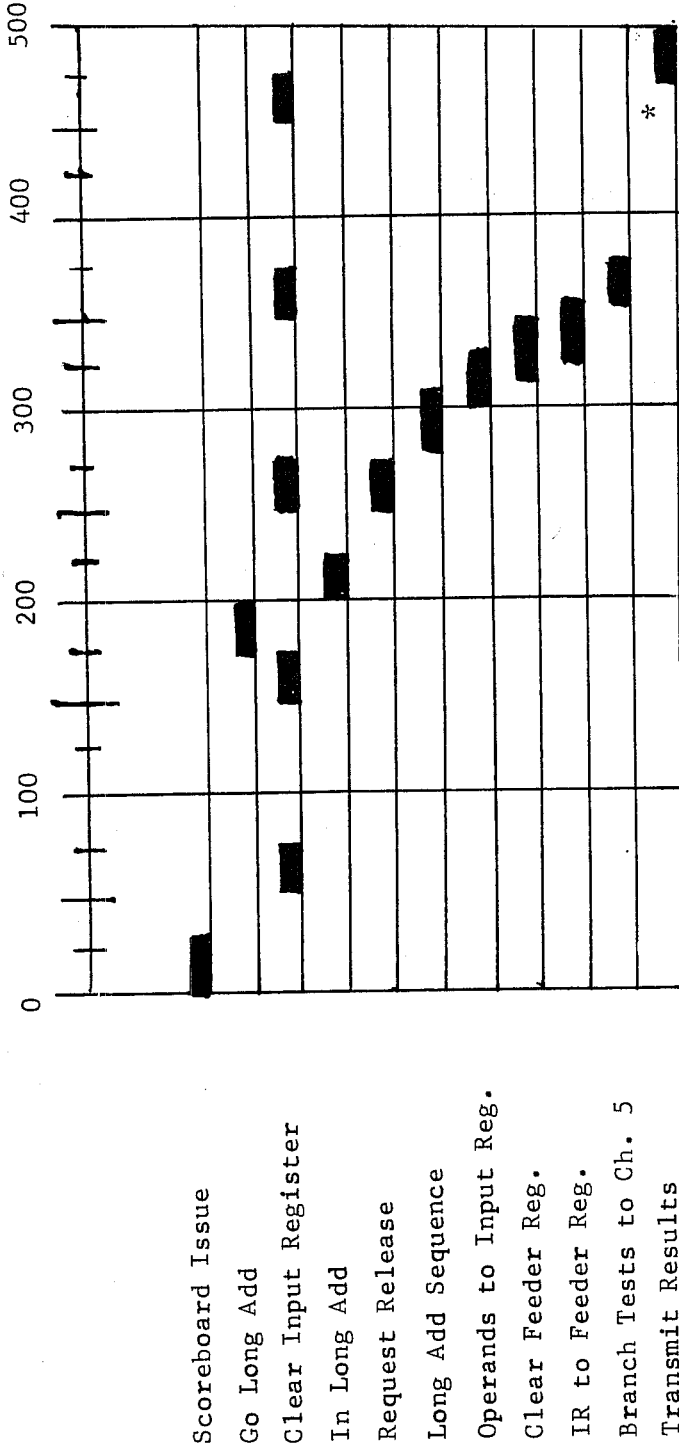
Figure 7.3-2



7.3-4 TIMING SEQUENCE

A timing chart for the Long Add Unit is shown in Figure 7.3-3. The following page explains the pulses shown on the chart. These two pages, in conjunction with the C.E. Diagrams (i.e. sheet 136) should explain quite fully the Long Add Sequence. The time has used (t_{000}) is the Scoreboard issue of the Long Add (36 or 37) or the Branch (03X) instruction.

LONG ALL UNIT - TIMING CHART



* Earliest possible time - No Result Register Conflict

Figure 7.3-3

t000 - Time reference - Scoreboard issue of 36, 37, or 03X instruction
t175 - The Go Long Add flip/flop (8H01) is set and starts the timing chain.
t050 - Chassis 8 input register is cleared each minor cycle with t40 (8L07-15)
t200 - Second flip/flop in timing chain (in Long Add) sets. (8J23, TP 1)
t250 - Request Release is sent to the scoreboard.
t285 - Third flip/flop in timing chain (8#28, TP 2) sets. Results in gating singla to chassis 5 (from 8H06,
27) for Branch test.
t300 - Operands (Xj and Xk) are received on chassis 8 (KT modules)
t310 - Feeders (FA modules) are cleared in preparation for receipt of operands.
t315 - Input registers (KT modules) are transferred to feeders (FA modules)
t350 - Control Bits resulting from branch tests are sent to the branch unit on chassis 5. (These are always
generated; used only if desired)
t475 - This is the earliest time possible to transmit the result of the Adder to register entry control. (If
a third order conflict occurs, the transmit will be later in multiples of 100; i.e. t575, t675, etc.)

7.3.5 ADDER

The Long Add Unit Adder forms the 60-bit integer sum ($X_j + X_k$) or difference ($X_j - X_k$) of the two source operands. To perform addition, both operands are sent to the feeders in true form. During subtraction, X_j is sent in true form and X_k is complemented. The decision to load the true or false value of X_k is made by the Long Add Plus flip/flop which is set only for the 36 instruction (See Section 7.3.3, Mode Bit).

As a preliminary to the explanation of the adder logic, the terminology used must be defined.

The Long Add Unit adder is said to be subtractive in nature. The distinction between additive and subtractive adders is made by looking at the result generated when adding complemented numbers. An additive adder will generate negative zero (all ones) when adding complemented numbers. In other words, it forms the quantity, $A + B$ (A and B being the source operands).

$$\begin{array}{r}
 \text{Additive} \\
 \text{Adder}
 \end{array}
 \qquad
 \begin{array}{r}
 542\text{---}673 \\
 235\text{---}104 \\
 \hline
 777\text{---}777
 \end{array}$$

A subtractive adder will generate positive zero when adding complemented values. In other words, it forms $A - (-B)$ or $A + B$ by the rules of algebra.

$$\begin{array}{r}
 \text{Subtractive} \\
 \text{Adder}
 \end{array}
 \qquad
 \begin{array}{r}
 542\text{---}673 \\
 +235\text{---}105 \\
 \hline
 \end{array}
 \qquad
 \begin{array}{l}
 = \\
 =
 \end{array}
 \qquad
 \begin{array}{r}
 542\text{---}673 \\
 -542\text{---}673 \\
 \hline
 000\text{---}000
 \end{array}$$

The Long Add adder generates positive zero when adding complemented quantities, although it is not accomplished by the "pencil and paper" method shown above. It is therefore a subtractive adder by definition.

In the discussion of the adder logic reference is made to the terms; Borrow, Satisfy, Enable, and Pass. These are defined using the subtractive approach, $A - (-B)$, as the criterion. Normally, when adding by complementing and subtracting the possible bit configurations are defined as in Figure 7.3-4A. In the case of the Long Add adder, the operands are contained in the feeders

<table style="margin-left: auto; margin-right: auto;"> <tr><td style="padding: 0 5px;">E</td><td style="padding: 0 5px;">B</td><td style="padding: 0 5px;">S</td><td style="padding: 0 5px;">E</td></tr> <tr><td style="padding: 0 5px;">A</td><td style="padding: 0 5px;">=</td><td style="padding: 0 5px;">0</td><td style="padding: 0 5px;">0</td></tr> <tr><td style="padding: 0 5px;">B</td><td style="padding: 0 5px;">=</td><td style="padding: 0 5px;">0</td><td style="padding: 0 5px;">1</td></tr> <tr><td style="padding: 0 5px;"></td><td style="padding: 0 5px;"></td><td style="padding: 0 5px;">1</td><td style="padding: 0 5px;">1</td></tr> <tr><td style="padding: 0 5px;"></td><td style="padding: 0 5px;"></td><td style="padding: 0 5px;">0</td><td style="padding: 0 5px;">1</td></tr> </table>	E	B	S	E	A	=	0	0	B	=	0	1			1	1			0	1	<table style="margin-left: auto; margin-right: auto;"> <tr><td style="padding: 0 5px;">E</td><td style="padding: 0 5px;">B</td><td style="padding: 0 5px;">S</td><td style="padding: 0 5px;">E</td></tr> <tr><td style="padding: 0 5px;">A</td><td style="padding: 0 5px;">=</td><td style="padding: 0 5px;">0</td><td style="padding: 0 5px;">0</td></tr> <tr><td style="padding: 0 5px;">B</td><td style="padding: 0 5px;">=</td><td style="padding: 0 5px;">1</td><td style="padding: 0 5px;">0</td></tr> <tr><td style="padding: 0 5px;"></td><td style="padding: 0 5px;"></td><td style="padding: 0 5px;">1</td><td style="padding: 0 5px;">0</td></tr> <tr><td style="padding: 0 5px;"></td><td style="padding: 0 5px;"></td><td style="padding: 0 5px;">1</td><td style="padding: 0 5px;">0</td></tr> </table>	E	B	S	E	A	=	0	0	B	=	1	0			1	0			1	0
E	B	S	E																																						
A	=	0	0																																						
B	=	0	1																																						
		1	1																																						
		0	1																																						
E	B	S	E																																						
A	=	0	0																																						
B	=	1	0																																						
		1	0																																						
		1	0																																						
(A)	(B)																																								

Figure 7.3-4

in true value, and the stages are labeled as in Figure 7.3-4B. Essentially, we are saying IF operand B was complemented, two ones (in true value) is a Satisfy, two zeros a Borrow, and any zero-one combination an Enable. A Pass has the same meaning as Not Satisfy.

The adder is divided into five 12-bit sections as follows:

59	48	47	36	35	24	23	12	11	0
Section	4	3	2	1	0				

Each section is further divided into four 3-bit groups as follows:

11	9	8	6	5	3	2	0
Group	3	2	1	0			

Since all sections and groups are logically similar, only one section (section 0) will be analyzed. Figure 7.3-5 is a logic diagram of section 0 and should be referenced in following this discussion.

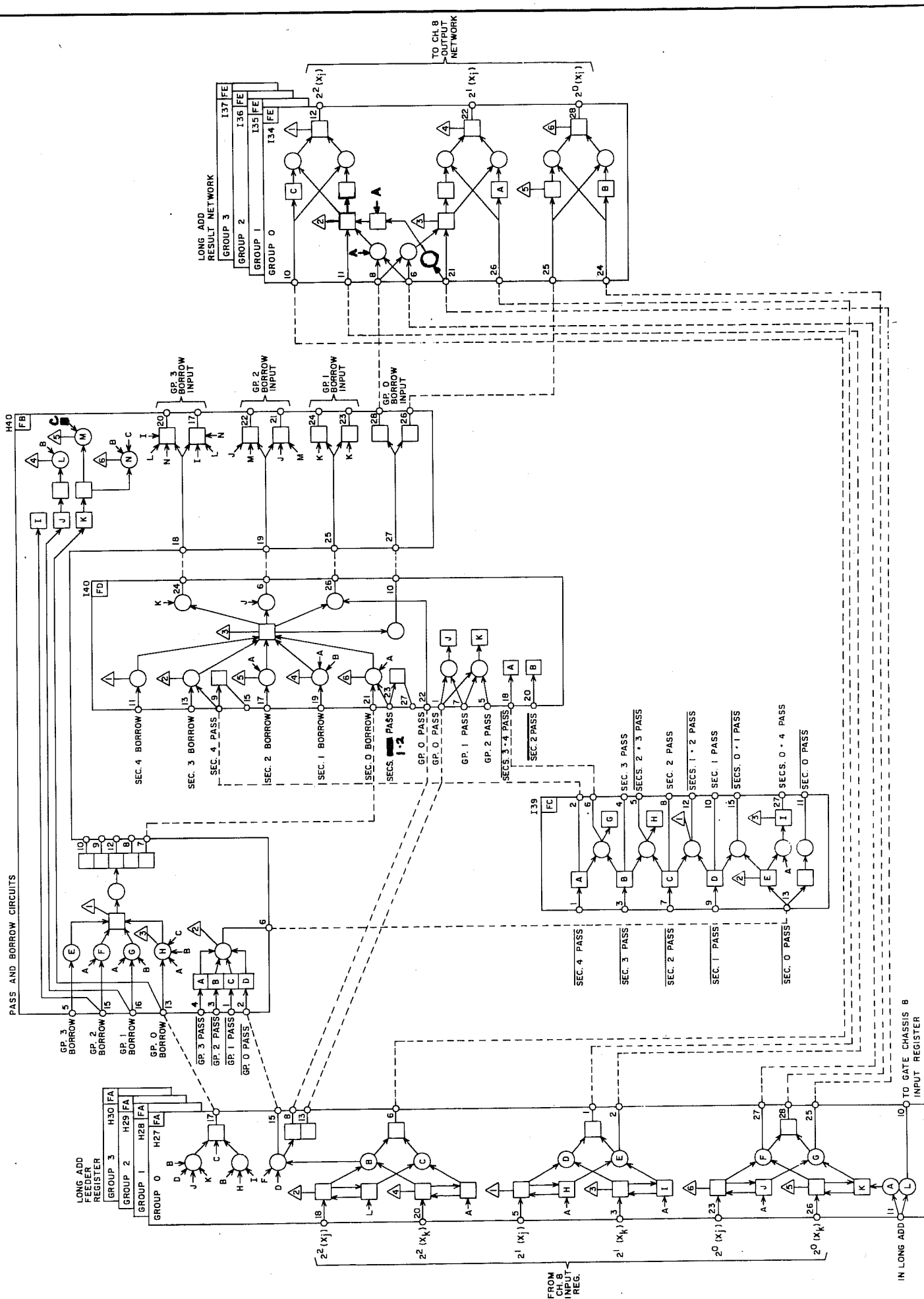


Figure 7.3-5

FA Modules

To the left of the diagram feeder registers for group 0 (Section 0) are shown. One FA module is required for each group; hence a total of 20 such modules are used. Each FA module checks for a group pass (pins 8, 13, and 15) and a group borrow (pin 17). These signals are sent to the Pass and Borrow summation circuits (FB, FC, and FD modules). Pins 6, 1 and 28 of the FA modules, when a logical 1, indicate a $\overline{\text{Enable}}$ (Equivalence) condition in the respective bit positions. Pins 2 and 5 indicate the $\overline{\text{Borrow}}$ state of bits 2^1 and 2^0 respectively. Pin 27 checks for the $\overline{\text{Satisfy}}$ (Pass) state of bit 2^0 .

FB and FD Modules

The Pass and Borrow circuits are used to summarize the pass and borrow conditions determined by the FA modules. Test Point 1 on the FB modules, for instance, checks for borrows generated in each of groups 0, 1, 2 and 3; checks for satisfies ($\overline{\text{Passes}}$) in the groups and will ultimately determine whether a borrow can be satisfied in this section, or whether it must be propagated to the other sections (from pin 7 of the FB to pin 21 of the FD). The FD module then compares the section borrows to the section passes ($\overline{\text{Satisfies}}$) and group passes. For example, pin 24 of I40 says (translated for a "zero"):

(groups 0, 1 and 2 of Section 0 contained no satisfies (term k))

AND

((Section 4 generated a borrow) OR (Section 3 generated a borrow and Section 4 could not satisfy) OR (Section 2 generated a borrow and Sections 3 and 4 contained no satisfies) OR (Section 1 generated a borrow and Section 0 generated a borrow and Sections 1, 2, 3 and 4 contained no satisfies))

To the right of the FB modules drawing borrows and passes for group 0, 1, 2, and 3 are combined to determine finally, which groups have borrow inputs and which do not. At this point, all possibilities for borrow inputs to any one group have been checked. It is now necessary to determine which stages within each group have borrow inputs and how this will affect the final result of each stage. This is the function of the FE modules.

FE Modules

Each FE module summarizes the borrow and enable (equivalence) conditions for one group (i.e, three stages). Thus, 20 FE modules are used.

Earlier, and Enable was defined as a 0, 1 or 1, 0 combination. With either of these bit configurations, a "one" should result (as the answer) if no borrow enters that stage, and a zero if a borrow does enter the stage:

In Boolean -

$(\text{Enable})(\overline{\text{Borrow}})$ to 1

$(\text{Enable})(\text{Borrow})$ to 0

Conversely, if an Enable is not present, equivalence (1, 1 or 0, 0) must exist in that stage. With no borrow the result should be "zero"; with a borrow the result should be "one". In Boolean -

$(\overline{\text{Enable}})(\text{Borrow})$ to 1

$(\overline{\text{Enable}})(\overline{\text{Borrow}})$ to 0

By analyzing the FE module of Figure 7.3-5 the above statements are confirmed. Figure 7.3-6 summarizes the possible combinations and the result obtained for any one stage, as determined by the FE modules.

Condition	Result
$(\text{Enable})(\overline{\text{Borrow}})$	1
$(\overline{\text{Enable}})(\text{Borrow})$	1
$(\text{Enable})(\text{Borrow})$	0
$(\overline{\text{Enable}})(\overline{\text{Borrow}})$	0

Figure 7.3-6

7.3.6 BRANCH TESTS

The Branch tests are used to condition the 03X series of jump instructions.

Four tests (Zero, Sign, Range, and Indefinite) are used as follows:

<u>Opcode</u>	<u>Name</u>	<u>Test</u>
030	Jump to K if $X_j = 0$	zero
031	Jump to K if $X_j \neq 0$	zero
032	Jump to K if $X_j \geq 0$	sign
033	Jump to K if $X_j \leq 0$	sign
034	Jump to K if X_j is in Range	range
035	Jump to K if X_j is out of Range	range
036	Jump to K if X_j is definite	indefinite
037	Jump to K if X_j is indefinite	indefinite

Zero Test

The zero test circuitry tests all 60-bits of the X_k input register (the content of the X register specified by the j portion of the instruction) for all zeros (positive zero) or all ones (negative zero). Any other bit configuration is considered a non-zero quantity.

During the discussion of the Zero Test Logic, refer to Figure 7.3-7. Recall that during 03X instructions, the Long Add Plus flip/flop (I19) is cleared and the complement of the X_k input register is sent to the X_k feeder. The X_j input register receives no input; consequently the X_j feeder contains all zeros.

For the positive zero test, a check for non equivalence between each bit of X_j and X_k is made. (i.e., FA module, pin 28) The non-equivalent conditions for all bit positions are ANDed (with sign = 0 or positive) on FE modules (i.e., J39). If any bit position is equivalent, the resulting "zero" out of the FE module will enable the transmitter on H06 (pin 22) to send an $X_j \neq 0$ signal to the Branch unit. Conversely, if all bit positions are non-equivalent, the transmitter is disabled; the absence of the $X_j \neq 0$ signal implies $X_j = 0$.

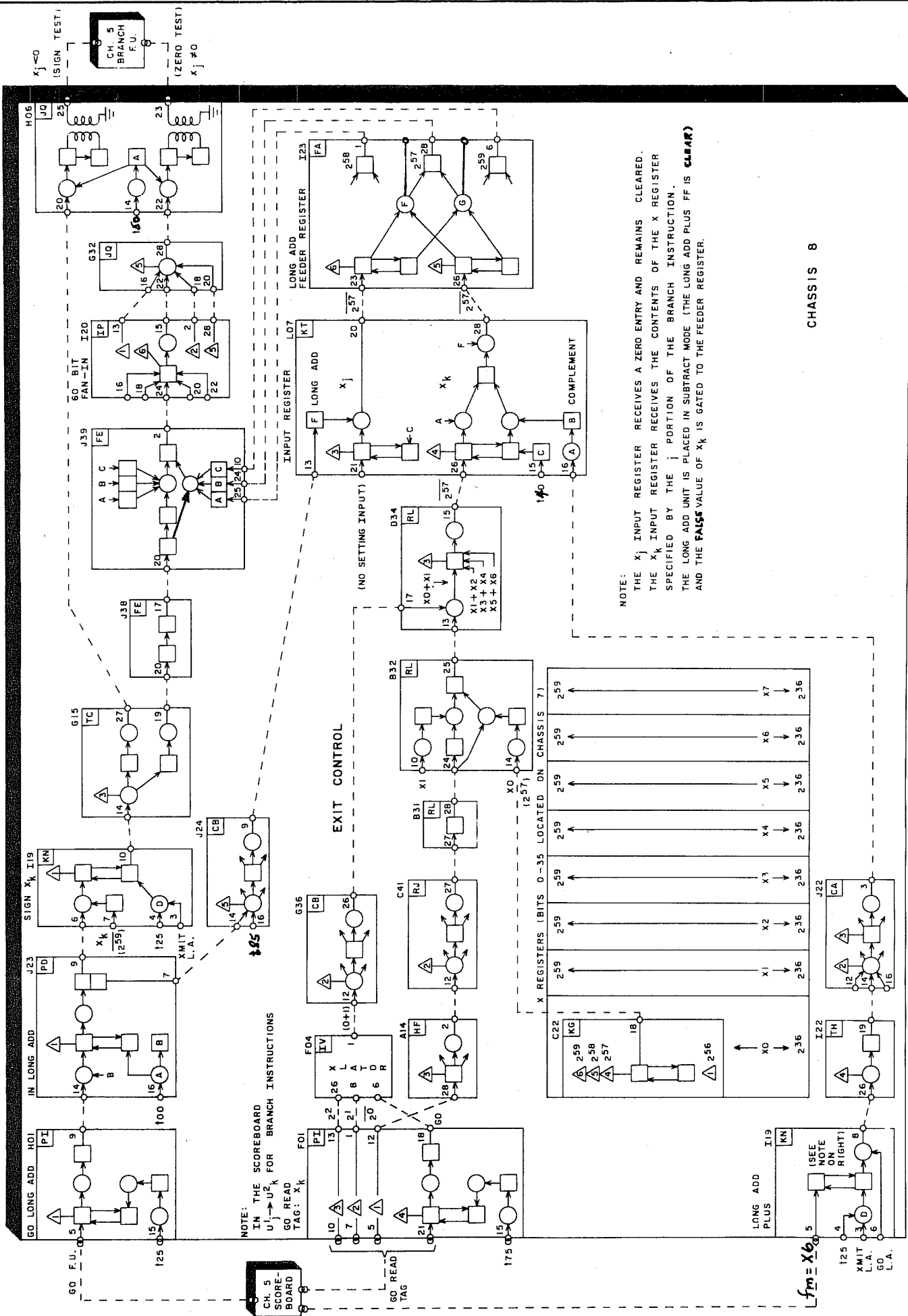
The Negative Zero test works in a similar manner. A check is made for all bit positions being equivalent. Any position resulting in non-equivalence will enable the $X_j \neq 0$ signal to be transmitted. The absence of the $X_j \neq 0$ signal will indicate to the branch unit that $X_j = 0$.

Sign Test

The sign test is a simple matter of checking bit position 2^{59} of the X_j register (on I19, test point 1). If set (indicating negative), a transmitter on H06 sends a signal to the Branch unit indicating the $X_j < 0$ condition. A positive sign (2^{59} cleared) disables the transmitter. The absence of the $X_j < 0$ signal implies $X_j \geq 0$. (See Figure 7.3-7)

Range Test

The range test checks bits 48-59 of X_j for negative or positive infinity (4000 or 3777). Since the lower 48 bits are ignored near overflow numbers are also considered out of range if the upper 12 bits equal 3777 or 4000.



NOTE:
 THE X_j INPUT REGISTER RECEIVES A ZERO ENTRY AND REMAINS CLEARED.
 THE X_k INPUT REGISTER RECEIVES THE CONTENTS OF THE X REGISTER SPECIFIED BY THE j PORTION OF THE BRANCH INSTRUCTION.
 THE LONG ADD UNIT IS PLACED IN SUBTRACT MODE (THE LONG ADD PLUS FF IS CLEAR) AND THE FALSE VALUE OF X_k IS GATED TO THE FEEDER REGISTER.

CHASSIS 8

Figure 7.3-7

Figure 7.3-8 shows the logic of the range/indefinite tests. The chassis 8 input registers and Long Add feeders are not used during these tests. Instead, the checks are made directly from the X registers via register Exit Control.

Two KS modules are utilized in making the negative and positive range tests.

K19 determines the state of bits 48-53. A "one" out of pin 20 indicates:

1. bits 48-53 are all "ones" and the sign is positive, $(0XX\ XXX\ 111\ 111_2)$
OR
2. bits 48-53 are all "zeros" and the sign is negative $(1XX\ XXX\ 000\ 000_2)$

The check is made by comparing each bit position with the sign (bit 59) of the register (terms E and F). K19, pin 20 then feeds pin 12 of K20, the second KS module, along with bits 54, 55, and 56. At test point 4, all the bit states are combined and the output of pin 20 indicates:

1. bits 48-57 are all "ones" and the sign is positive $(0X1\ 111\ 111\ 111_2)$
OR
2. bits 48-57 are all "zeros" and the sign is negative $(1X0\ 000\ 000\ 000_2)$

The output of K20, pin 20, is returned to K19, pin 19. At this point the circuit looks at bits 58 and 59 in combination to determine the existence of $\dagger 1777_8$ (indefinite) or $\dagger 3777_8$ (out of range) or neither. The translation for K19, pin 14 $((58)(\overline{59}) \vee (\overline{58})(59))$ is ANDed with pin 19 and both pins equaling a one imply an out of range condition (3777 or 4000). This is indicated by a "one" on pin 13 which enables setting test point 2 on F17 via D20 and G17. On the following t50, the "Xj out of range" signal is transmitted to the Branch unit from H06.

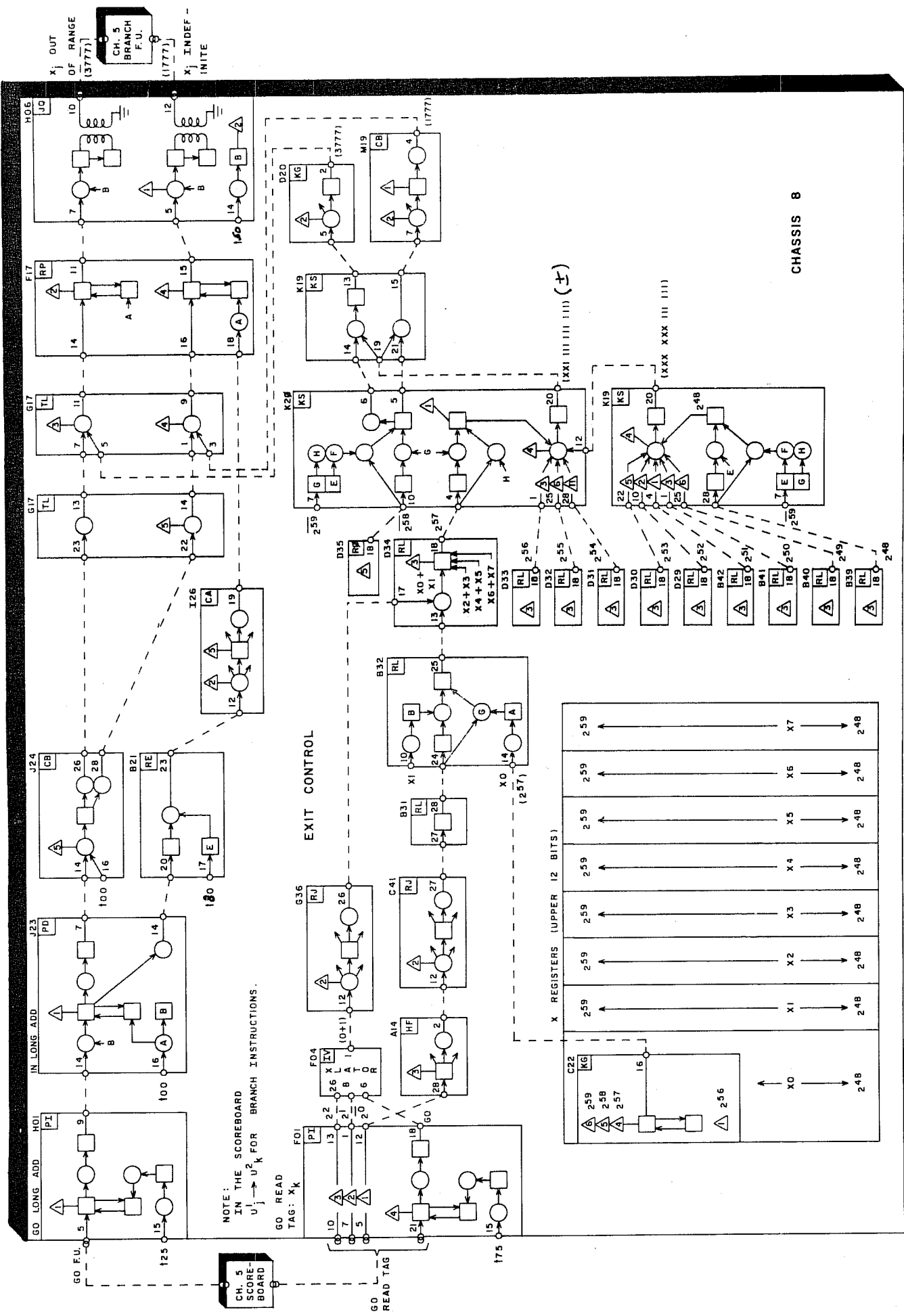


Figure 7.3-8

Indefinite Test

This test utilizes the same circuitry as the range test (Figure 7.3-8) except that the K19, pin 19 translation (OX1 111 111 111₂ or 1X0 000 000 000₂) is ANDed with K19, pin 21. When equal to a "1" this pin indicates $(\overline{58})(\overline{59})$ or (58)(59). If both pins 19 and 21 equal "1's", pin 15 will be a zero indicating an indefinite condition (1777 or 6000). This output enables setting test point 4 on F17 via M19 and G17, which in turn enables the H06 transmitter to send the "Xj indefinite" singla to the Branch unit.

Synchronization of the Branch test results from the Long Add Unit with the sequence of the Branch Unit is accomplished by extending the Long Add timing sequence (Figure 7.3-9). A transmitter on H06 is conditioned by the "In Long Add" flip/flop and sends a signal (called "Long Add Sequence to Ch. 5") to the Branch Unit. This results in a signa called "Auxilliary Functional Unit Release" which enables the branch sequence to continue after making the In Stack/Out Stack tests. (Refer to Branch Unit, Section 7.8)

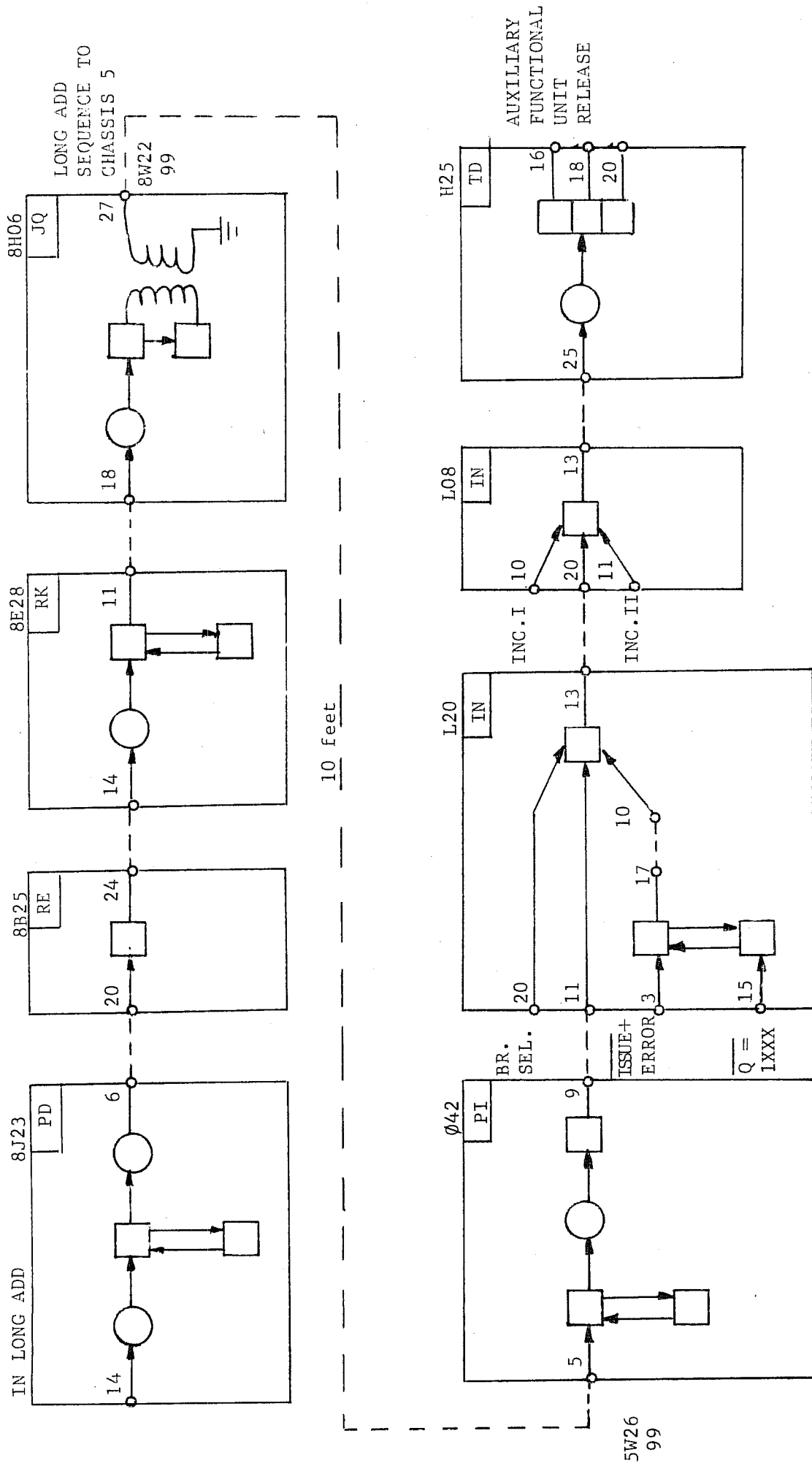


Figure 7.3-9

SECTION 7.7

INCREMENT

Functional Units

INCREMENT FUNCTIONAL UNITS

7.7.1 INTRODUCTION

The Increment Functional Units are 18-bit, fixed point arithmetic units which perform these general functions.

1. Indexing
2. Reading and Storing Operands
3. Conditional Branch Tests

One's complement addition and subtraction of 18-bit operands is performed in accomplishing the Indexing, Read Operand, and Store Operand functions. Operands may be selected from A registers, B registers, X registers (the truncated, lower 18 bits), or the K portion of a 30 bit instruction.

The following instructions are classified as indexing instructions: (They are discussed in detail in section 7.7.2)

5X0 (where X = 0-7) The result of the arithmetic process specified by octal "X" is stored in A register zero (A0).

6X Instructions (where X = 0-7) The result of the arithmetic process specified by octal "X" is stored in any one of B registers 1-7. (B0 is a constant all-zero word; if specified as a result register, the result is lost).

7X Instructions (where X = 0-7) The result of the arithmetic process specified by octal "X" is stored in any one of X registers 0-7. Since an 18-bit result is stored in a 60-bit register, the sign of the result (bit 2^{17}) is extended to the upper 42 bits of the X register.

02 Instruction The result of the arithmetic process (in this case, $B_i + K$) specifies a jump address. The 02 (unconditional jump) is always out of the stack. Therefore the result is sent to the P register and an RNI is initiated.

The following instructions may incorporate the indexing function in their operations, but the end result of executing these opcodes is to read or store an operand. They are therefore classified separately as Read and Store operand instructions.

5X1 - 5X5 (where $X = 0-7$) The result of the arithmetic process specified by octal, "X", is stored in the A register specified by the i digit (1-5). The result is also sent to memory as an operand address. A memory read cycle is made and a 60-bit word is read from memory into the X register specified by the i digit (X1 - X5).

5X6 - 5X7 (where $X = 0-7$) The result of the arithmetic process specified by octal "X" is stored in the A register specified by the i digit (6 or 7). The result is also sent to memory as a store address for an operand. A memory write cycle is initiated and a 60-bit word from the X register specified by the i octal is stored in the memory location specified by the result.

The following instructions are classified as Conditional Branch Test instructions. These opcodes cause both Branch and Increment functional units to start at the same. While Branch performs the In Stack/Out Stack tests (see Section 7.8) the Increment unit selected compares two 18-bit operands. The results of the tests are returned to the Branch unit where they are used in determining whether or not the branch condition specified

was met. These instructions also are discussed in greater detail in Section 7.7.2.

The 04-07 instructions will jump to location K if the specified condition is met.

- 04 - $B_i = B_j$
- 05 - $B_i \neq B_j$
- 06 - $B_i < B_j$
- 07 - $B_i > B_j$

Logically there are two Increment functional units. Although they share a common arithmetic section, their control portions are separate - but interlocked in special circumstances. (see Figure 7.3-1). Because of the duplexed control circuits, two increment instructions in sequence normally will not cause a functional unit conflict. (A special case does exist where a unit conflict will occur. This is explained in later paragraphs).

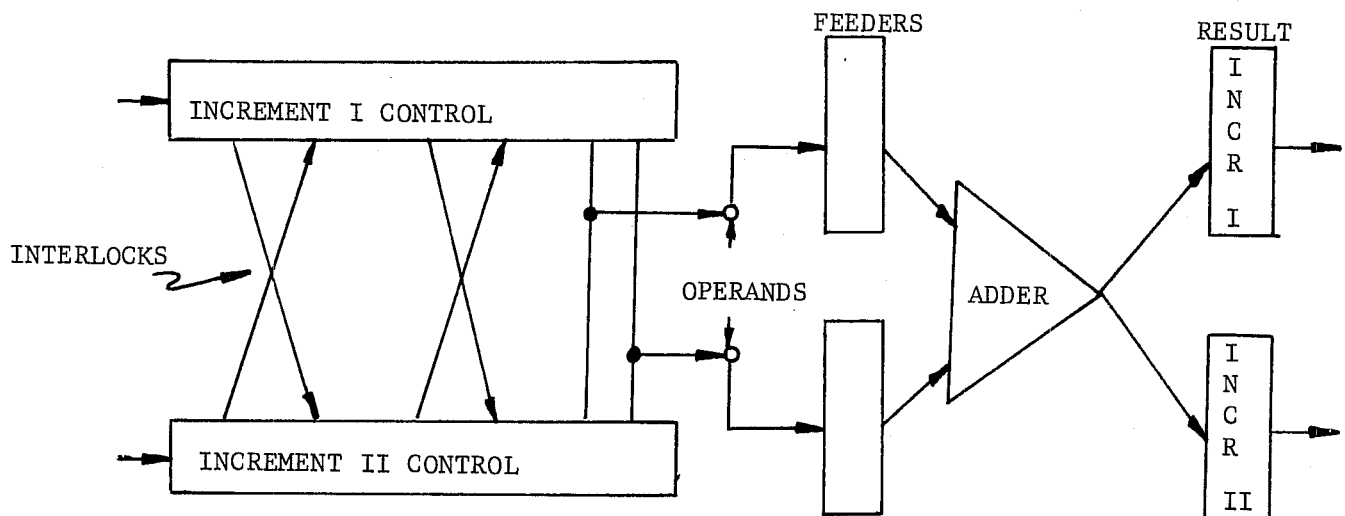


Figure 7.7-1

7.7.2 INSTRUCTION LIST/DATA FLOW

The instruction set for the increment units includes 29 opcodes classified in four groups:

- a) 50 - 57 - Result register is A_i
- b) 60 - 67 - Result register is B_i
- c) 70 - 77 - Result register is X_i
- d) 02, 04-07 - Branch instructions

The 5X, 6X, and 7X use the same source operands for corresponding values of the octal digit, X, but the instruction groups differ in two respects; 1) the result register specified and 2) the 5X series causes operand read (if $i = 1-5$) and write (if $i = 6 + 7$) memory cycles.

5X Instructions:

50	SUM of A_j and K to A_i	(30 bits)
51	SUM of B_j and K to A_i	(30 bits)
52	SUM of X_j and K to A_i	(30 bits)
53	SUM of X_j and B_k to A_i	(15 bits)
54	SUM of A_j and B_k to A_i	(15 bits)
55	DIFFERENCE of A_j and B_k to A_i	(15 bits)
56	SUM of B_j and B_k to A_i	(15 bits)
57	DIFFERENCE of B_j and B_k and A_i	(15 bits)

These instructions perform one's complement addition and subtraction of 18-bit operands and store an 18-bit result in the address (A) register designated by the i octal. Note that the j operand may be selected from any one of the X, B, or A registers. The second operand may be any one of the B registers or the 18-bit constant, K.

Depending on the value of octal i , an operand read or write cycle may be initiated by the 5X instructions.

If $i = 0$, no memory reference is made. The result is simply sent to A register zero (A0).

If $i = 1, 2, 3, 4, \text{ or } 5$, an operand read memory cycle is initiated. This will cause a 60-bit word to be read from the memory location specified by the result of the operation, into the X register specified by octal i (1-5). Thus, two result registers are used, A_i and X_i , by the 5 X 1 - 5 X 5 opcodes.

If $i = 6 \text{ or } 7$, an operand write memory cycle is initiated. This will cause a 60 bit operand from the X register specified by octal i (6 or 7) to be stored in the memory location specified by the result of the operation. Thus, the result register is A_i , and X_i is, in a sense, a source register for memory, for the 5 X 6 and 5 X 7 opcodes.

Data Flow (Refer to Figure 7.7-2)

Upon issuing the increment instruction to the scoreboard, the 18-bit constant K is gated from the R register to the Result register (I or II) of the selected increment unit by the K to Incr. I or II gate. After resolving any second order conflicts which may have occurred, the selected Increment unit is sent a "GO" signal which starts its timing sequence. The operands (determined by the m octal of f_m) are sent from register Exist control to the Input registers (feeders) of the 18 bit adder. If $f_m = 50, 51, \text{ or } 52$ the 18 bit constant K is sent to the feeder for operand two by the Enter K , Incr I or II gate. If K is not used ($f_m \neq 50, 51, \text{ or } 52$) B_k is used

INCREMENT FUNCTIONAL UNITS - BLOCK DIAGRAM

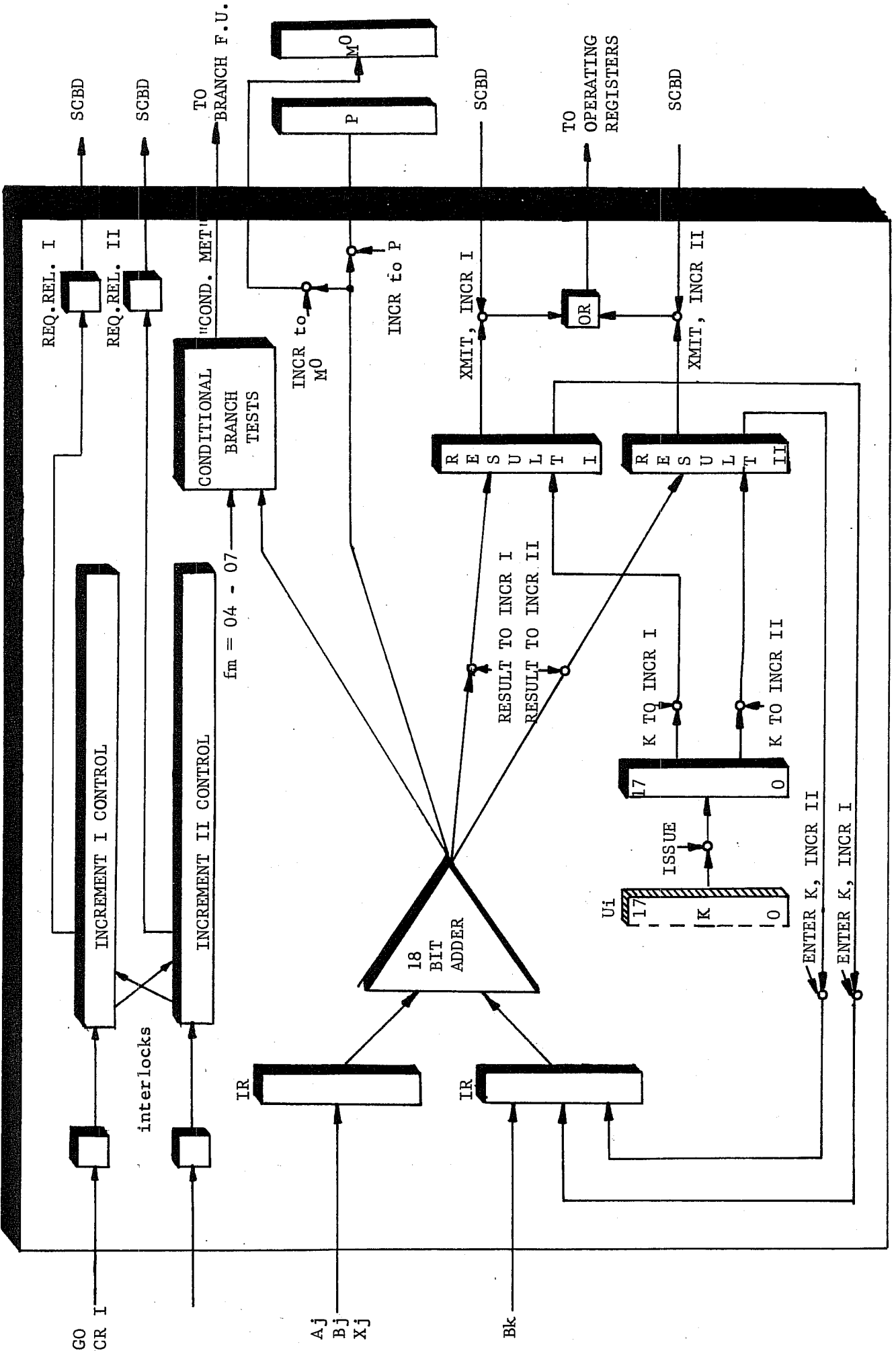


Figure 7.7-2

as the second operand. If $f_m = 55$ or 57 , B_k is complemented into the feeder register and a difference is subsequently formed.

The arithmetic result is unconditionally sent to either the Result II or Result I register, (depending upon which Unit was selected). When the "transmit" signal is received from the scoreboard, the result is sent to the designated (by the i octal) A register. If $i = 1$ through 7 , the result is also sent to the M^0 register of the Stunt Box and priority is requested by setting the Enter Central flip-flop. If i was equal to $1-5$, a memory read cycle is initiated and a 60-bit result will be sent from memory to the X register designated by the i octal. If X was equal to 6 or 7 , a write memory cycle is initiated and a 60-bit operand from X_6 or X_7 is stored in Central Memory.

6X Instructions

60	SUM of A_j and K to B_i	(30 bits)
61	SUM of B_j and K to B_i	(30 bits)
62	SUM of X_j and K to B_i	(30 bits)
63	SUM of X_j and B_k to B_i	(15 bits)
64	SUM of A_j and B_k to B_i	(15 bits)
65	DIFFERENCE of A_j and B_k to B_i	(15 bits)
66	SUM of B_j and B_k to B_i	(15 bits)
67	DIFFERENCE of B_j and B_k to B_i	(15 bits)

These instructions perform one's complement addition and subtraction of 18-bit operands and store an 18 bit result in the B register designated by the i octal. The j operand may be selected from any one of the X, B, or A registers. The second operand may be any one of the B registers or the 18 bit constant, K.

Data Flow (Refer to Figure 7.7-2)

With the exceptions of initiating operand read or store memory cycles, and storing the result in a flow for 6X instructions is the same as the 5X series. Once again, the operand combinations are selected by the m portion of fm and the result is gated to the selected B register by the "transmit" signal.

7X Instructions

70	SUM of Aj and K to Xi	(30 bits)
71	SUM of Bj and K to Xi	(30 bits)
72	SUM of Xj and K to Xi	(30 bits)
73	SUM of Xj and Bk to Xi	(15 bits)
74	SUM of Aj and Bk to Xi	(15 bits)
75	DIFFERENCE of Aj and Bk to Xi	(15 bits)
76	SUM of Bj and Bk to Xi	(15 bits)
77	DIFFERENCE of Bj and Bk to Xi	(15 bits)

These instructions perform one's complement addition and subtraction of 18-bit operands and store the 18-bit result with sign (2^{17}) extended, in the X register designated by the i octal. The j operand may be selected from any one of the X, B, or A registers. The second operand may be any one of the B registers or the 18-bit constant, K.

Data Flow (Refer to Figure 7.7-2)

With the exception of the result register selected (X instead of B), data flow is the same as the 6X instructions. Operand combinations are selected by the m octal of fm and the result is gated to the selected X register by the "transmit" signal. The sign of the result (bit 2^{17}) is extended to bits 18-60 of the X register by using a fan-out.

02, 04-07 Branch instructions

02 GO TO K + Bi (30 bits)

This instruction adds the contents of B register i to K and branches to the location specified by the sum. Addition is performed in modulus $2^{18}-1$. The branch address is K when $B_i = B_0$.

Data Flow (Refer to Figure 7.7-2)

The operand from B_j^* and 18-bit constant K are placed in the feeder registers. B_j is sent from Exit Control and K from the Result I or II registers with the "Enter K" gate. The operands are added and the "Incr to P" gate is enabled, sending the sum to the P register. The result is also sent to Result I or II register (unconditionally), but since a "transmit" is not received from the scoreboard, the result is not sent to the operating register. The result is also sent to M^0 and from there will be sent to M^1 , when stunt box priority is granted, to initiate the RNI.

*Recall that on OX instructions, U_i^1 and j are sent to U_j^2 and k respectively. B_j is thus designated by B_i of the original opcode and B_k by B_j .

04-07 Branch Instructions

- 04 GO TO K if $B_i = B_j$ (30 bits)
- 05 GO TO K if $B_i \neq B_j$ (30 bits)
- 06 GO TO K if $B_i \geq B_j$ (30 bits)
- 07 GO TO K if $B_i < B_j$ (30 bits)

These instructions test an 18-bit word in register B_i against an 18-bit word in register B_j (both words signed quantities) for the condition specified and branch to address K on a successful test.

Data Flow (Refer to Figure 7.7-2)

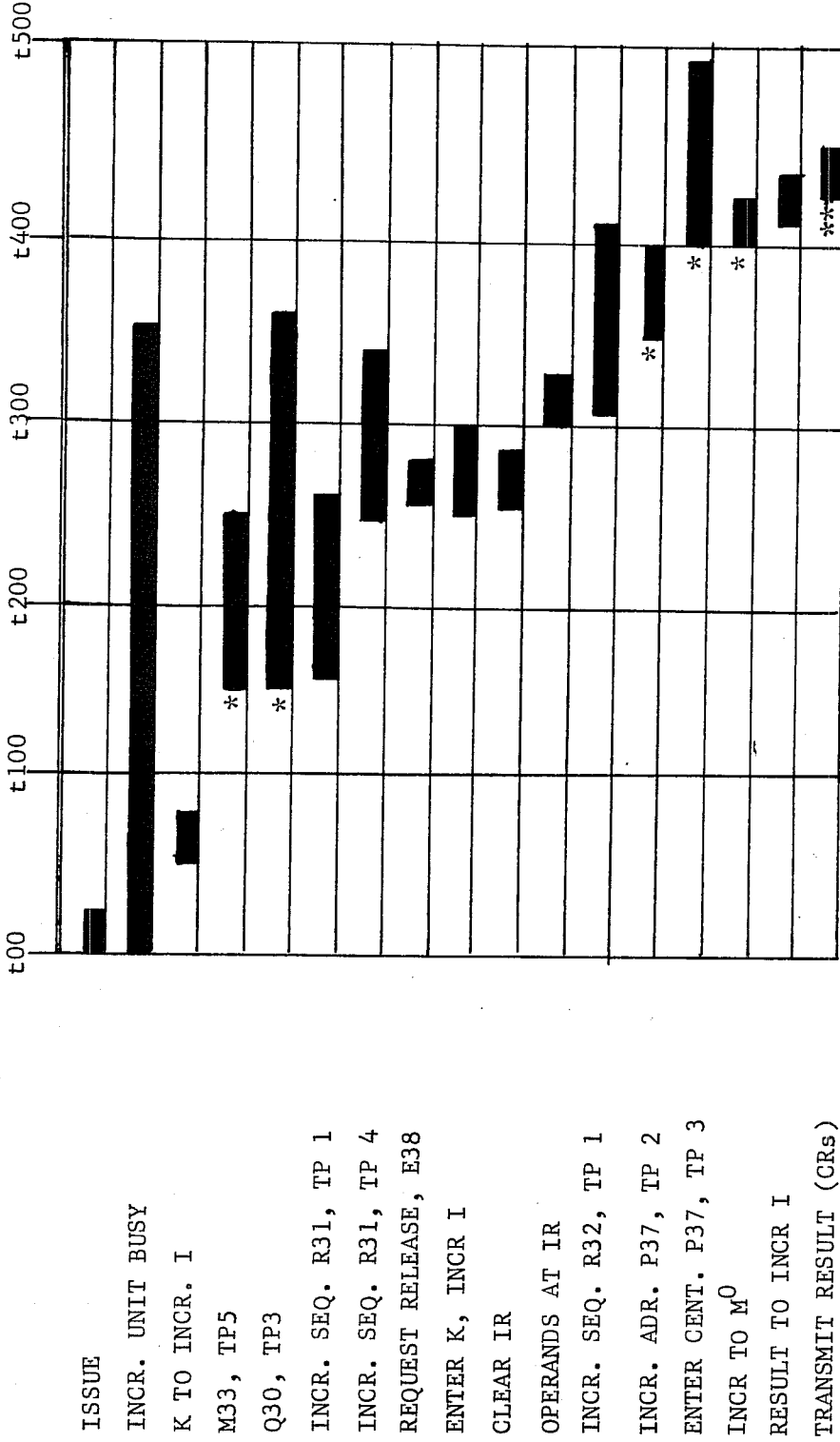
Operands B_j^* and B_k^* are sent to the Increment Unit feeder registers. For the conditional tests of equality (04 and 05) a bit by bit equivalence check is made in the result network of the adder. If all bits positions are equivalent, the 04 jump may be enabled. If any bit position is not equivalent, the 05 jump may be enabled. For the conditional test of magnitude (06 and 07), the sign bit of the one's complement difference ($B_i - B_j$) is examined.

If it is zero, $B_i \geq B_j$; if a one, $B_i < B_j$.^{*} In all four cases, the result of the tests are combined with the opcode translation to generate the "Condition Met" signal (the absence of "Condition Met" implies "Condition Not Met") which is sent to the Branch functional unit to enable the branch sequence to continue.

*Recall that on OX instructions, U^1_i and j are sent to U^2_j and k respectively. B_j is thus designated by B_i of the original opcode and B_k by B_j .

^{*}In certain cases, overflow will yield an incorrect result. For example, when $B_i=300000$ and $B_j=577777$, the result, 500000, carries a sign bit of 1, indicating $B_i < B_j$ when, in fact, $B_i > B_j$.

INCREMENT TIMING SEQUENCE



*Applicable only to Increment Read and Writes (5X1 -5X7)

**Earliest possible time - no third order conflicts.

Figure 7.7-3

ISSUE: The scoreboard issue of the Increment instruction is used as the time reference for the chart.

INCR. UNIT BUSY: The unit busy flip/flop is set for approximately 350 nsec. This unit may be reselected at t400.

K to INCR I: The content of the R register is sent to the Increment Result Register (I or II).

M33, TP 5: Set if a 5X1 - 5X7 (C.M. Read or Write) opcode is executed. It will be used to disable two memory operations to take place at the same time (See section 7.7.4).

Q30, TP 3: Set if a 5X1 - 5X7 (CM Read or Write) opcode is executed. It will be used to set the Increment Address flip/flop (P37, TP 2).

INCR. SEQ., R31, TP 1: The first flip/flop of the Increment timing chain. It is set with (GO INCREMENT) and (NO READ OR WRITE IN PROGRESS). (See section 7.7.4) Used to enable Request Release.

INCR. SEQ., R31, TP 4: The second flip/flop of the Increment timing chain.

REQUEST RELEASE: Sent to the Scoreboard's All Clear network to resolve any third order conflicts which may exist.

ENTER K, INCR I: Enabled by setting Q21, TP 4 on the time 25 following R31, TP 1 if the following condition exists: $fm = (5X + 6X + 7X)(X0 + X1 + X2) + 02$.

CLEAR IR: Clears the input registers in preparation for loading operands.

OPERANDS AT IR: Operands are received from Register Exit Control on the JA modules.

INCR SEQ., R32, TP 1: The third flip/flop of the Increment timing chain (Used to gate results from the address output network).

INCR ADR. P37, TP 2: Set if a 5X1-5X7 (CM Read or Write) opcode is executed. Used to gate Increment result to M⁰

ENTER CENTRAL P37, TP3: Set due to setting Incr. Adr. Flip/flop. Request priority to enter address into M¹

from M^0 . (only during 5X1 - 5X7 instructions)

INCREMENT TO M^0 : The gate which sends Increment result to M^0 (during 5X1 - 5X7 instructions)

RESULT TO INCR. I: Gates adder output network into Increment result register (I or II)

TRANSMIT RESULT: Result is transmitted to result registers with t25 on CR modules (transmitters).

7.7.4 ADDER CONTROL

General Information

When all Read flags are set, the "Go Increment" signal starts the control sequence of the Increment Unit selected. The adder is common to both Increment units, but as has been mentioned, the control sequences are separate for each unit. The control sequence is a timing chain which accomplishes the following (Refer to section 7.7.3 for specific timing information).

1. Transmits Request Release to scoreboard.
2. Clears adder feeder registers (IRs)
3. The "Go Increment" is ANDed with an fm translation and in the case of a $(5X + 6X + 7X)(X0 + X1 + X2)$ or 02 instruction, gates K to the input register. (K, the lower 18 bits of U^2 are unconditionally sent to the R register on every issue. If an Increment Unit is selected, R is gated to Increment Register I or II).
4. The "Go Increment" is ANDed with another fm translation and for $(5X + 6X + 7X)(X5 + X7)$ or 04 - 07 instructions (difference or branch tests), Bk or K is complemented out of the Input Register into the adder.
5. "Go Increment" clears all Read flags.
6. Gates the result from the adder into the Increment I or II register. (Figure 7.7-4 summarizes the sources of operands and destinations of results).

INSTRUCTION	RESULT	DESTINATION
02	$K + B_i$	M^0 and P registers
04 - 07	Condition Met or Not Met	Branch Unit
5X, $i = 1-7$	$(A+B+X)_j$ plus $(B_k + K)$	M^0 and A_i Registers
5X, $i = 0$	$(A+B+X)_j$ plus $(B_k + K)$	A_i register
6X	$(A+B+X)_j$ plus $(B_k + K)$	B_i register
7X	$(A+B+X)_j$ plus $(B_k + K)$	X_i register

Figure 7.7-4

The results to M^0 and P registers are transmitted from the result network of the adder. The results to the operating registers are first sent to the Increment I or II registers from the result network.

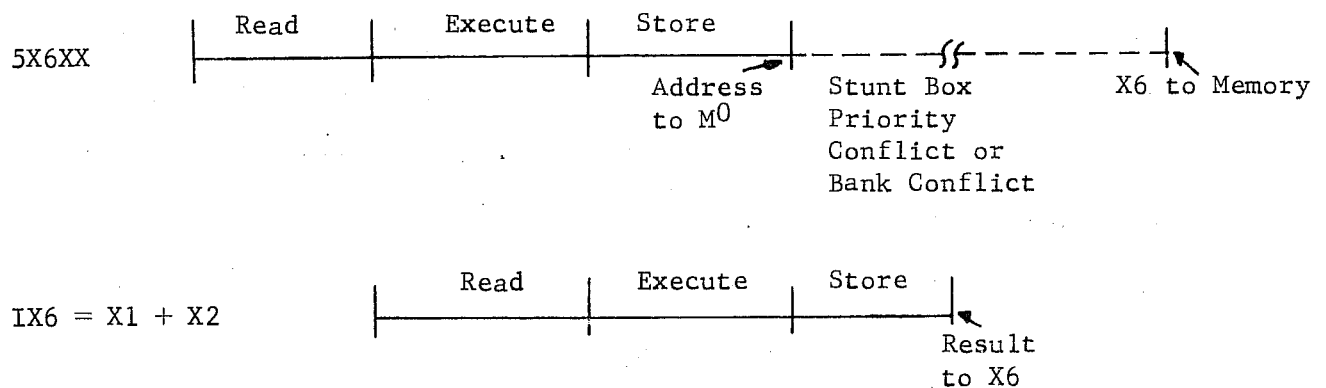
Increment First Order Conflicts (Special Cases): (5X)($i = 1-5$)

When an increment instruction (5X)($i = 1-5$) is issued to the scoreboard a reservation is made for the X and A register involved. A is reserved with an Increment Unit code (01 or 02) and X with a Memory to X code (11 - 15). Thus before the scoreboard issue is enabled, both X and A registers must be free. On Modules E26 and E27, both A_i and X_i reservations will be checked (term T), and the scoreboard issue occurs only if both reservations are cleared.

The A_i and X_i reservations are cleared by separate gates, A_i is cleared upon "Release" of the Increment unit (Request Release and All Clear). X_i is cleared only when the address sent to memory is accepted. Logically, the hopper tag is translated (11 - 15) and is ANDed with the Accept for that tag.

(56)(i = 6 + 7)

In the case of a 5X instruction where $i = 6$ or 7 , the A register (6 or 7) is the result register of the Increment unit, while X6 or X7 may be considered "source registers" for memory. When considered in this light, it seems that the A register should be reserved, but the X register need not be reserved (it is not a result register). As far as the XBA reservation list is concerned, the A register is reserved (code = 01 or 02), but the X register is not. If the store operation was delayed (by second order conflicts or memory priority) and if no other circuitry was involved in handling operand store instructions, it would be possible for a subsequent instruction to specify X6 or X7 as a result register and to change X6 or X7 before storing the previous content in memory. Hence, the wrong operand would be stored. Study the following example:



The result of the Long Add instruction is stored instead of the previous content of X6.

To eliminate this problem, two flip/flops are used (refer to Figure 7.7-5). TP 6 on L01 is cleared (via pin 15) whenever an Increment Write is issued with bit 2^0 of the i octal equal to zero. This implies a 5X6 instruction. L01 pin 17 feeds the result register reservation logic and whenever a result register, X6 or A6 is desired, a first order conflict results. TP 6 on L02 serves the same function for an Increment write of X7 (5X7 instruction). Thus, a "pseudo-reservation" of X7 or X6 takes place during Increment Write operations. The reservations are cleared by ANDing an "Accept" with translations of the lower three bits of the hopper tag (X6 or X7). (The upper octal need not be translated since, with the exception of exchange jump hopper tags, no other tags use a second octal of 6 or 7. Hence a tag of 56 or 57 is implied).

The implications of this special case are as follows: 1) The "pseudo reservations" made on modules L01 and L02 will cause a first order conflict with any instruction requiring X6 or X7 (respectively) as a result register. A requirement of A6 or A7 as a result register will cause a first order conflict in the normal fashion - by translating A6 and A7 reservations in the XBA designator list for "not equal to zero". Of course, both of these cases will stop issue until the reservations are cleared. 2) Second order conflicts will occur only if A6 or A7 is required as a source register by a subsequent instruction. Since the X6 and X7 reservations are not made in the XBA reservation list by 5X6 and 5X7 instruction, second order conflicts with subsequent instructions wishing to read X6 or X7 will not occur as a result of Memory Write reservations. (Other instructions may have reserved X6 or X7 and therefore cause a conflict).

SPECIAL CASE: (5X6 or 5X7)

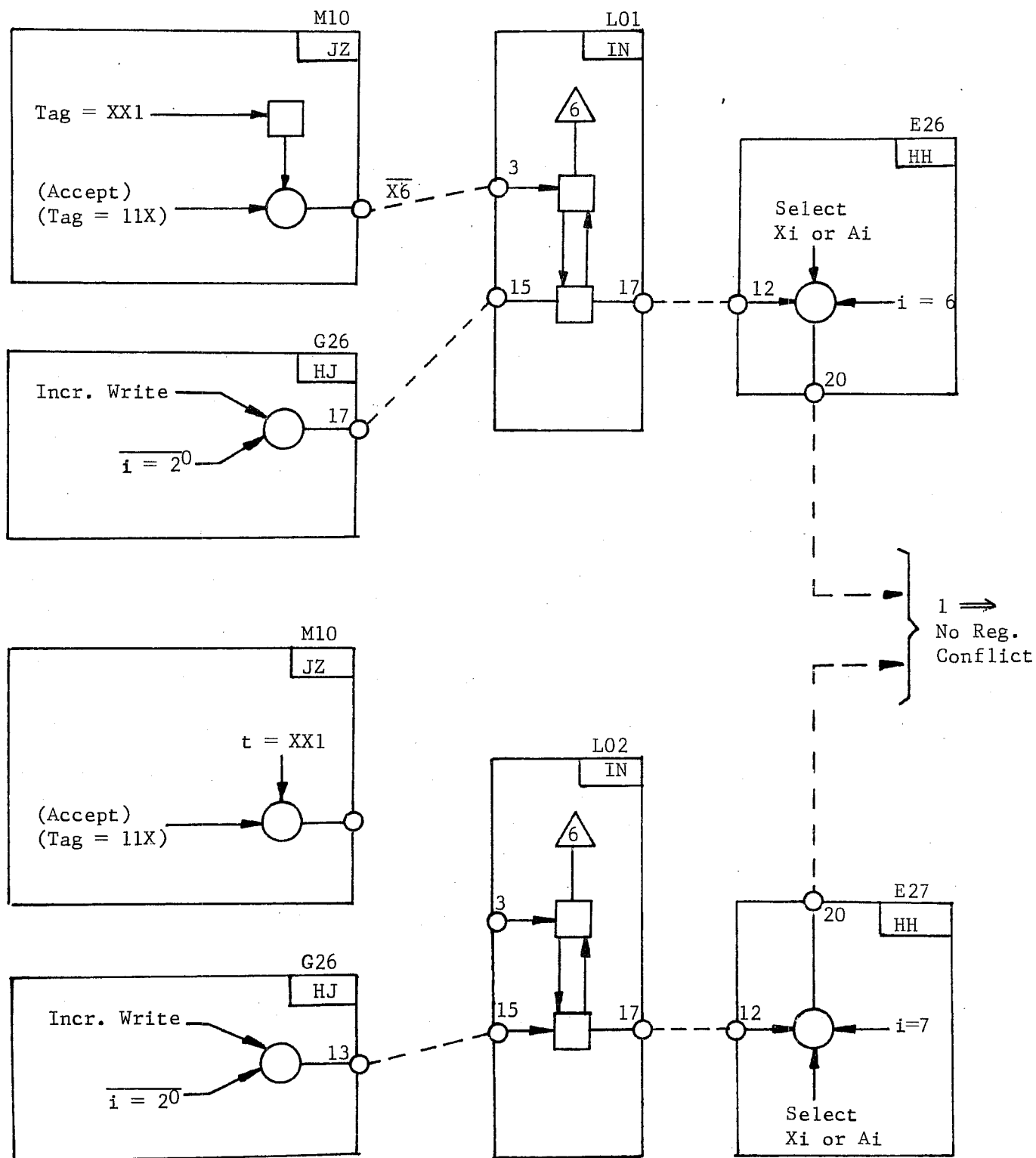


Figure 7.7-5

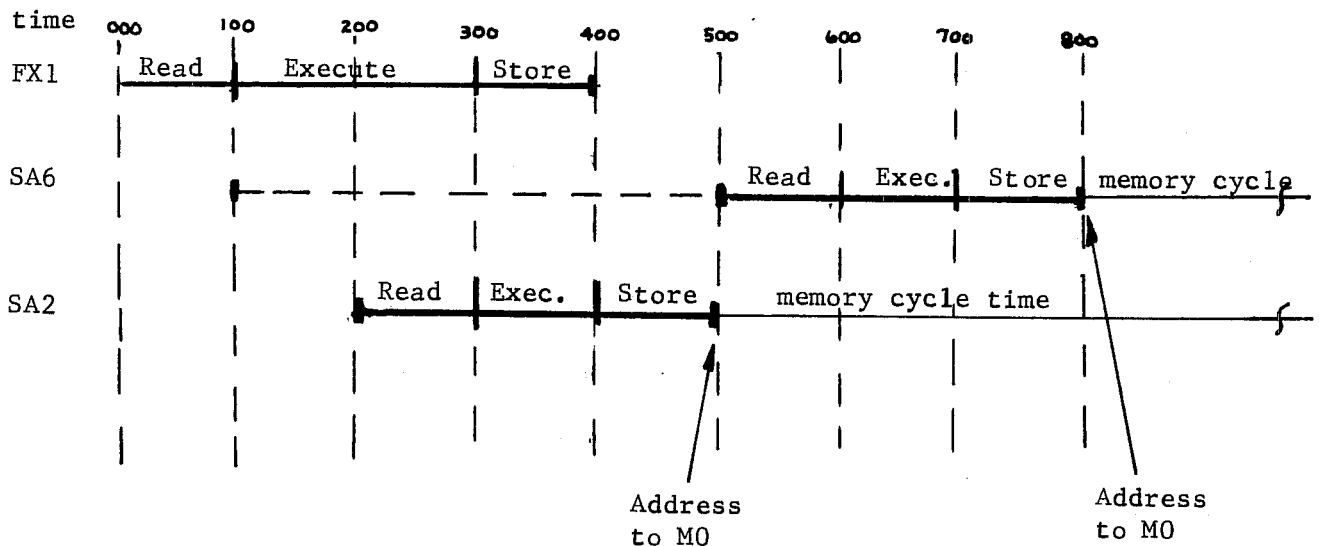
Mixed Memory Modes

Another situation will cause a third type of "unique" first order conflict. This is the case in which an Increment instruction of one memory mode (Read or Write) is coded subsequent to an Increment instruction of the other mode. A problem exists in preventing the two memory references from getting out of sequence. This could happen if, for instance, the first Increment Unit was held up due to a second order conflict. This problem is significant only if the two instructions reference the same address in central memory. It would be possible then, to read a location before storing the new operand (assuming the store was programmed before the read, and the mix-up did occur). Storing before reading is the second possibility. Study the following example.

FX1 = X2 + X3 (30123)
 SA6 = X1 + K (5261KKKKKK)
 SA2 = B1 + K (5121KKKKKK)

Assume that $X1 + K = B1 + K$

The timing looks as follows:



Conclusion: Although the store instruction (SA6) was coded before the read instruction (SA2) the read address was sent to the stunt box first. Hence, the content of the memory location will be read into X2 before storing X6 in the location. The programmer obviously intended to store before reading.

Fortunately, this erroneous operation cannot occur. It is prevented on modules G26 and G27, the Increment Unit busy circuits (Figures 7.7-6 and 7.7-13) shows the Unit Select and Busy logic for Increment Unit I. Increment II is handled on G27 in a similar manner; therefore only Increment I will be discussed.

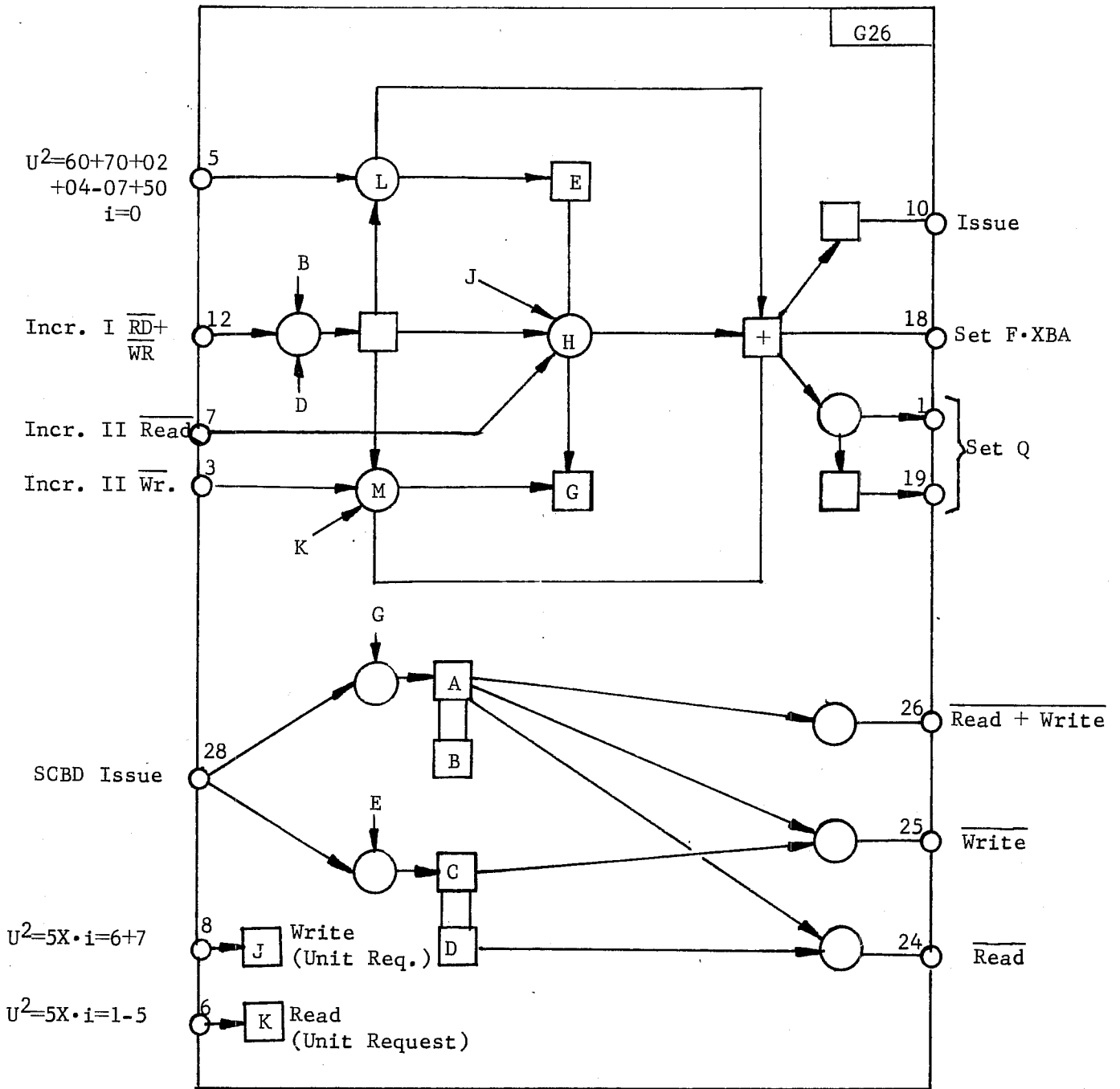
The Unit Request flip/flops (set on the U2 issue) have inputs on pins 6, 8, and 5. Term J indicates a Write unit request and term K a Read unit request. In order to generate the issue for a memory read request (term K) the AND gate, term M, must have all ones in. The translation for a zero out of M is: (Memory Read Request)(Incr. I Not Busy)(Incr. II Write). Hence, if Increment II is doing a Write and this request is for a Read, subsequent issues are disabled until Increment II finishes its Write operation (its Unit Busy flip/flops, A/B and C/D, are cleared with "release" of the unit).

Similarly, if the request is for a Write (term J) term H must have all ones in to enable the issue. The translation for a zero out of H is: (Memory Write Request)(Increment Not Busy)(Incr. II Read). Hence, if Increment II is doing a Read and this request is for a Write, subsequent issues are disabled until Increment II finishes its Read operation. (Its Unit Busy flip/flops, A/B and C/D are cleared with "release" of the unit).

Thus, attempting a memory operation of one mode while the other mode is in process will cause a first order conflict.

The flow charts (figures 7.7-7 through 7.7-10) illustrate the operation of the Increment select logic.

Select Control Increment Unit I



Flip-Flop		Incr. Selected For:
A/B	C/D	
Set	Clear	READ
Set	Set	WRITE
Clear	Set	INCREMENT
Clear	Clear	FREE

Figure 7.7-6

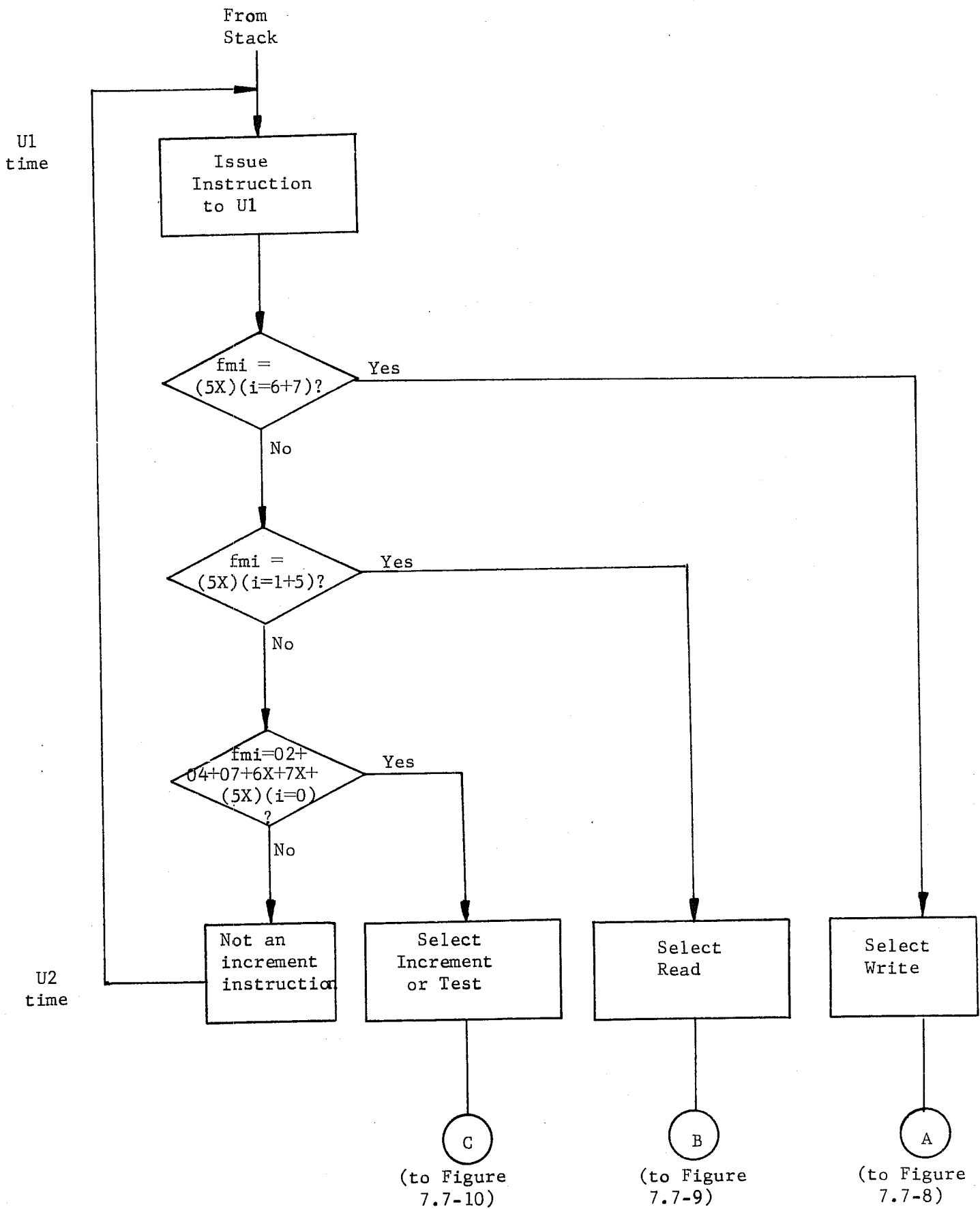
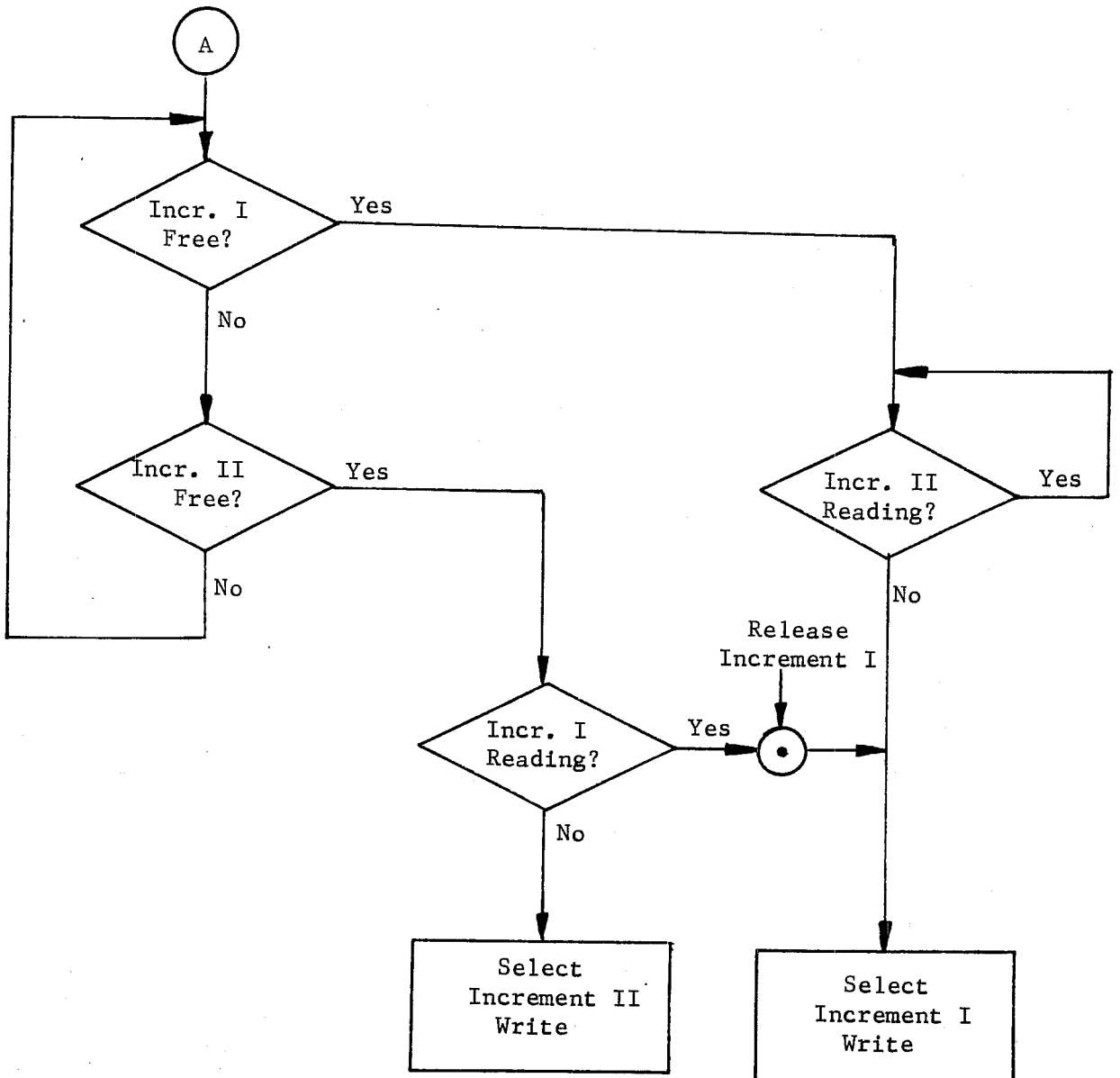
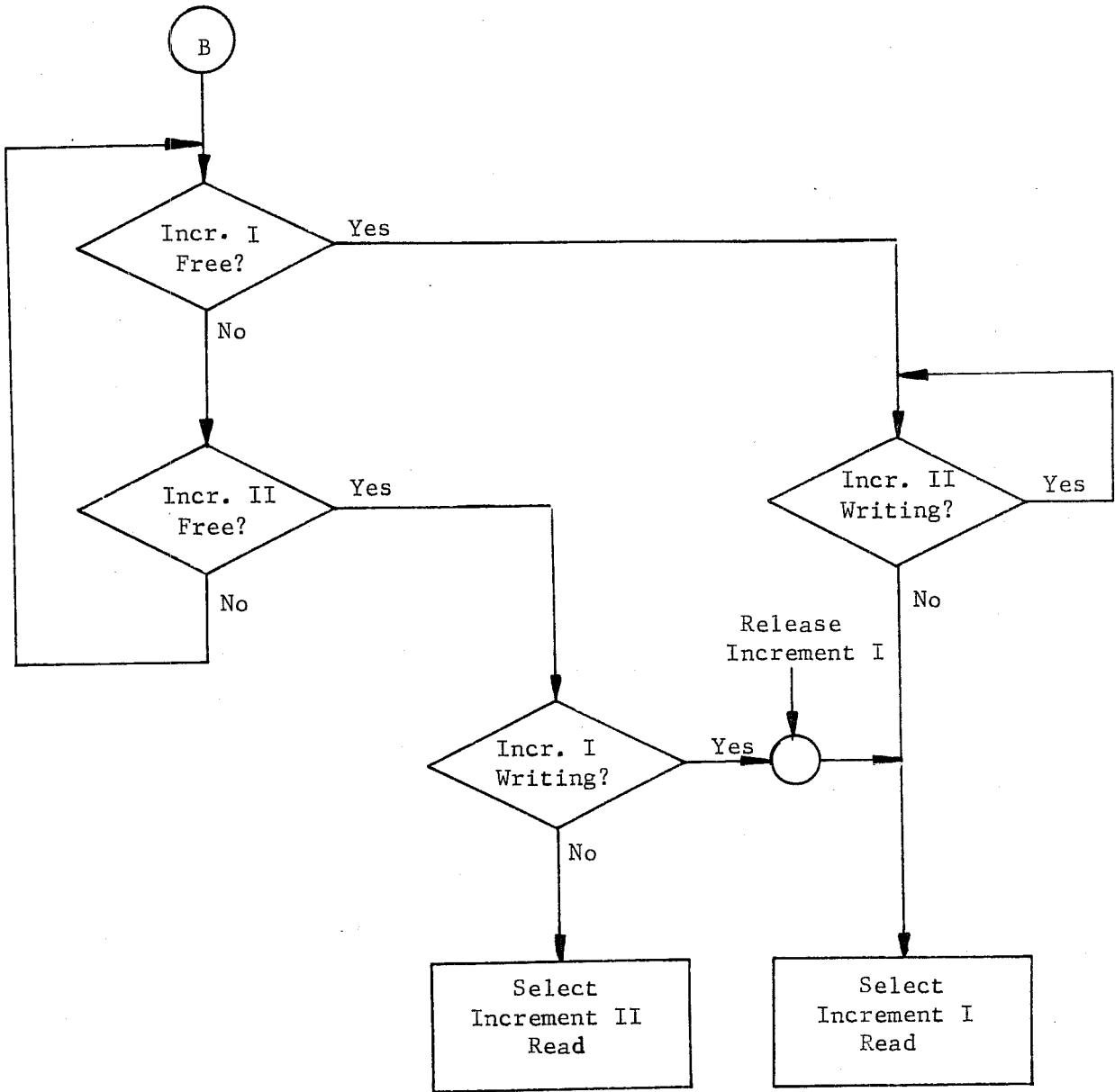


Figure 7.7-7



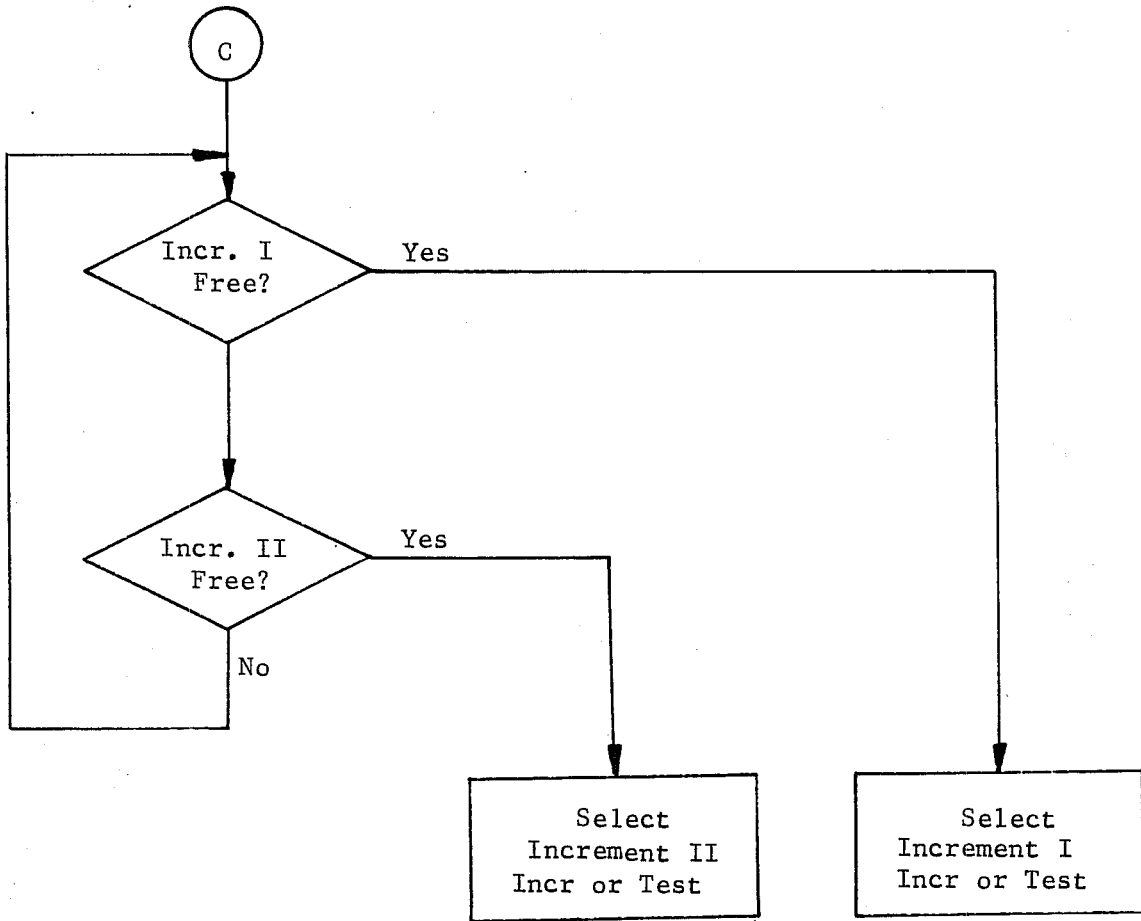
Select Increment Write

Figure 7.7-8



Select Increment Read

Figure 7.7-9



Select Increment or Test

Figure 7.7-10

Increment Second Order Conflict (Special Case)

A special case of a second order conflict arises in the Increment units, and the case again applies only to memory mode instructions (5X1 - 5X7). To illustrate the problem, assume that Increment unit I was issued a Read operand instruction. It has generated the address which is now sitting in the output network of the Increment adder. Assume also, that the address cannot be sent to M0 because M0 contains some other central address. This could occur, for instance, if the content of P or a previous operand address was sent to M0 and central priority (M0 to M1) is not granted because the Hopper contains an unaccepted address. Now, assume that another operand read instruction (5X1 - 5X5) is issued. (It will be issued if Increment II is not busy since Increment I is also handling a Read mode instruction). If the second Read instruction were allowed to start, the address generated by Increment I would be destroyed, because the adder is a static network - once the operands are loaded into the feeders, the result appears at the output within 80 nanoseconds. Thus, the logic must prevent reading the operands for Increment II until Increment I sends its address to M0. (Of course, the same problem exists if the Increment units were reversed or with two instructions of the Write mode.)

This situation is resolved with the logic contained on module M33. (Refer to Figures 7.7-11, 7.7-12, and 7.7-13). For the purpose of explanation, an example is discussed.

Assume that in a program, two increment instructions separated by several other instructions appear. Both of these instructions are of the type which cause a memory reference and are of the same memory mode (i.e. both read or both write). The first is issued to the scoreboard and begins to execute

via Increment Unit I. Since it is a memory mode instruction, M33, TP5 will be set (Figure 7.7-11). Within 300 nanoseconds the address is generated and will be sent into the result register for Increment I. Note also, that the Increment address flip/flop on P37 (Figure 7.7-13) is set later in the Increment sequence. This in turn causes Enter Central to be set which requests Stunt Box priority. Assume at this point that an RNI address is setting in M0 and stunt box priority 2 is not granted (due to priority 1, Read/Write tag conflicts, etc.). As a consequence, the RNI address remains in M0 and the operand address remains at the output network of the adder.*

Assume that by this time the program has progressed and now attempts to issue the second Increment instruction of our example. Since Increment II is free and the memory modes are the same, a scoreboard issue is generated.

Hence, Increment II and result registers are reserved in the normal fashion. If the source registers required are not reserved, the Increment II read flags will be set and a logical "one" will appear on pin 4 of M33 ("Go Incr. II").

In order to start the Increment II sequence, term "D" is the clear side of TP5, which was set by the first Increment instruction of the example. TP5 is cleared when term "F" is a logical "zero", in other words when all inputs to "F" are ones. The following translation yields a "Zero" out of term "F":

$$(540)(\overline{\text{Prog.Addr.}})(\overline{\text{Incr.Addr.}})(\overline{\text{Inch}})(\overline{\text{Branch}})(\overline{\text{Enter Cent.}} + M0 \text{ to } M1)$$

*If no third order conflicts exist Increment I may be released and the result sent to the specified A register. Nevertheless, the Increment to M0 gate may still be delayed.

START INCREMENT SEQUENCE

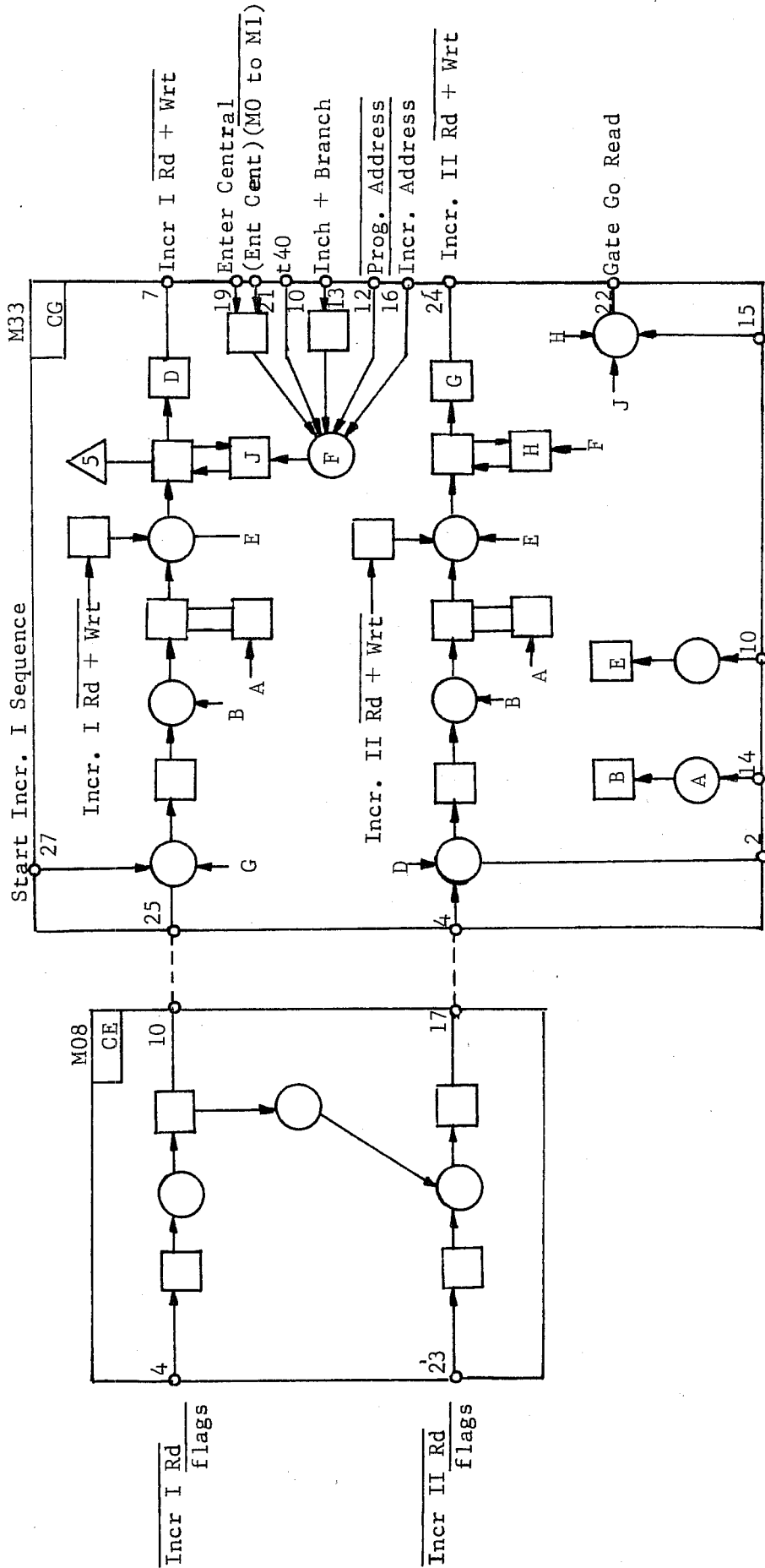


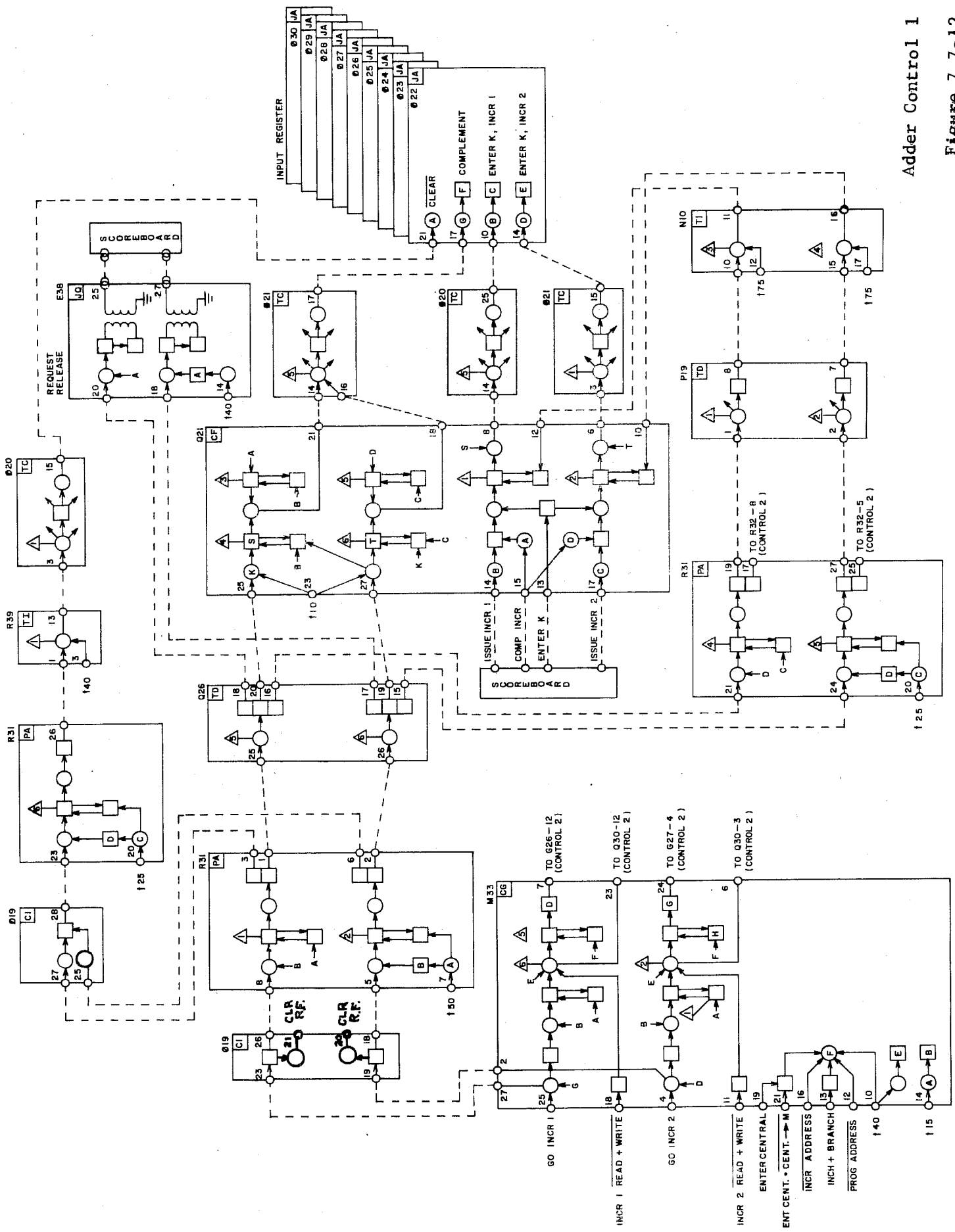
Figure 7.7-11

Starting Increment I for the first instruction caused the setting of the Increment Address and Enter Central flip/flops. The Increment Address is cleared when the signal, Increment to M0 is generated. This signal requires the following conditions:

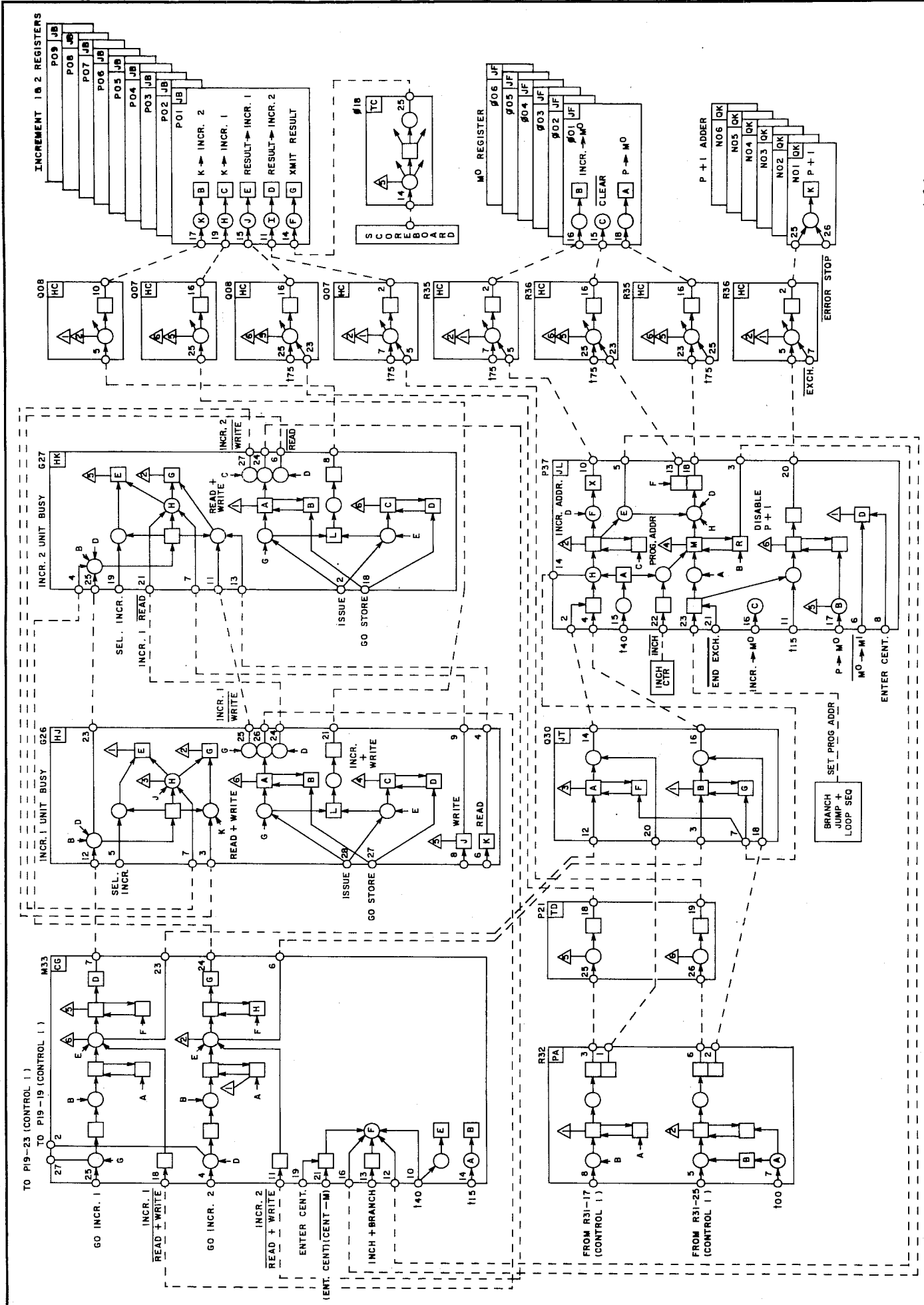
$$\text{Incr} \rightarrow \text{M0} \leftarrow (\text{Incr. Addr.})(\text{Enter Central} + \text{M0 to M1})$$

When priority is finally granted for the RNI address, the "M0 to M1" gate will be generated. This then will enable "Incr to M0" which in turn clears the Increment Address flip/flop. "M0 to M1" will also clear the Enter Central flip/flop. All conditions required by term "F" are now met. M33, TP5 will be cleared and the starting of Increment II is enabled. Note, that making the AND gate for starting Increment II also allows the clearing of that unit's Read flags (M33, pin 2 feeds Q19, pin 19 where the "clear RFs" is fanned out.) (Figure 7.7-12)

This explanation also applies to starting the Increment I sequence. If the result Increment II can not be sent to M0, Increment I may not start until term "G" is a one, implying that Incr to M0 did occur. These cases then are special second order conflicts applicable only to operand read or write instructions.



Adder Control 1
Figure 7.7-12



Adder Control 2

Figure 7.7-13

7.7.5 ADDER

General Information

The Increment units' 18-bit adder is subtractive in nature, but because of the logic configuration of the input register (JA modules) the true value of Bk or K is gated for Add instructions and the one complement value for Subtract. The true value of the first operand (Aj, Xj or Bj) is always used.

In explaining the adder logic, the following definitions apply:

$$\begin{array}{rcl} \text{Borrow} = & 0 & \\ & \underline{0} & \\ \text{Satisfy} = & 1 & \\ & \underline{1} & \\ \text{Enable} = & 1 & \text{or} & 0 \\ & \underline{0} & & \underline{1} \end{array}$$

Since the adder is subtractive in nature, the definitions presented are based upon the process of complementing and subtracting to add. In other words, if the adder did subtract to perform addition the second operand would have to be complemented. The following table relates the two processes:

True Operands (if adding to add)	Second Operand Complemented (if subtracting to add)	Condition of Stage
0 0	0 1	Borrow
1 1	1 0	Satisfy
0 1	0 0	Enable
1 0	1 1	Enable

Figure 7.7-14

Hence, even though the true values of the operands are used in the feeders during addition, the condition of a stage is defined with the "subtract to add" process in mind. A "Pass" has the same meaning as "Not Satisfy".

Pencil and Paper Method

The "Pencil and Paper" method of adding will be discussed before analyzing the adder logic. Assume that the following binary numbers are to be added.

$$\begin{array}{r}
 \begin{array}{c} \text{EAC} \\ \curvearrowright \end{array} \\
 \begin{array}{r}
 10010101 \\
 10100011 \\
 \hline
 00111000 \\
 1 \\
 \hline
 00111001 = \text{Sum}
 \end{array}
 \end{array}$$

By simple binary addition, the sum should be as shown.

The pencil and paper method which simulates the machine addition process is summarized as follows:

1. Label each stage according to its condition (Borrow, Satisfy, Enable)
2. Perform an "exclusive OR" between the source operands. In other words, for any stage containing an "enable" write a "one". (defined by Figure 7.7-14)
3. For each stage that has a Borrow input, write a "one".
4. Perform an "equivalence" between the first "exclusive OR" and the list of borrow inputs.

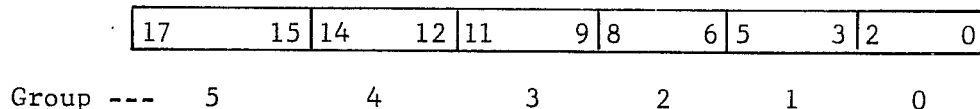
Example:

$ \begin{array}{r} \begin{array}{cccccc} \swarrow & \swarrow & \swarrow & \swarrow & & \\ \text{S} & \text{B} & \text{E} & \text{E} & \text{B} & \text{E} & \text{E} & \text{S} \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ \hline 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{array} \end{array} $	<ol style="list-style-type: none"> 1. label stages operand #1 operand #2 2. "exclusive OR" 3. borrows into stage 4. "Equivalence"
--	---

After completion of step 4, the correct sum has been generated.

Adder Logic Analysis

The adder is divided into six groups, each containing three bits as follows:



Since the groups all operate similarly, only group zero is analyzed in detail. Figure 7.7-15 is a logic diagram of Group 0 and should be referenced during the following discussion.

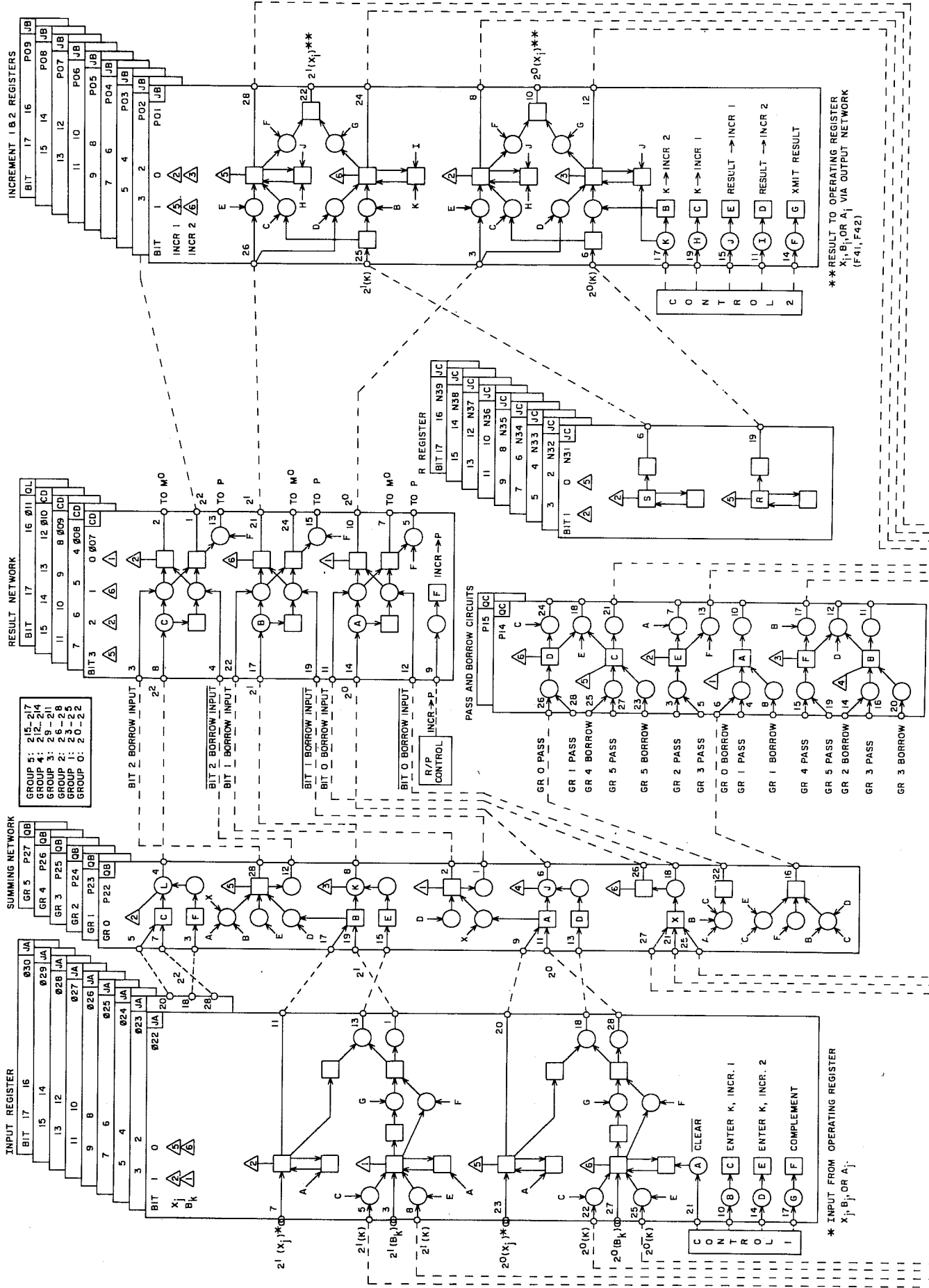
JA Modules

The JA modules hold the feeder registers. Each stage on these modules sends three signals to the QB modules (summing network). Stage 2^0 , for instance, has outputs from pins 20, 18, and 28. Respectively, these pins translate as j , j or k , and k . Note, that during difference or branch test operations the complement gate (term F) will be a one, enabling the complement of operand k . In this case, the three outputs are j , j or \bar{k} , and \bar{k} . Since the rest of the difference process is the same as addition, only addition will be discussed.

QB Modules

With reference to the "pencil and paper" example, the QB modules perform four main functions.

1. Perform the exclusive OR between the source operands. (Step 2 of the pencil and paper example) The output of test point #4, for instance, states that stage 2^0 does not contain an enable (Not "exclusive OR")
2. Check for a borrow leaving each stage. Using Stage 2^0 as an example,



** RESULT TO OPERATING REGISTER
 X₁, B₁ OR A₁ VIA OUTPUT NETWORK
 (F41, F42)

* INPUT FROM OPERATING REGISTER
 X₁, B₁ OR A₁

Figure 7.7-15

test point #1 (pin 2) states that stage zero has a borrow output. In other words, stage 1 has a borrow input. Pin 1 says no borrow into stage 1 (Test point #1 inverted). This is step 3 of the pencil and paper example.

3. Determine whether or not this group contains all passes (no satisfies). Pin 22 states, for instance, that group zero is all passes.
4. Checks for a borrow leaving the group. Pin 16, for instance, states that a borrow does leave group zero.

Note: The results of steps 3 and 4 are ultimately used to determine the existence of an End Around Borrow. (See QC module discussion)

CD Modules (and QL)

The CD modules perform step 4 of the pencil and paper example - equivalence between borrows and enables. For example, pin 10 (the 2^0 sum) translates is follows:

$$(\text{Borrow})(\text{Enable}) + \overline{(\text{Borrow})}\overline{(\text{Enable})}$$

The following table summarizes the possible combinations at CD modules and the resulting sum.

Condition	Sum
(Borrow)(Enable)	1
$\overline{(\text{Borrow})}\overline{(\text{Enable})}$	1
$\overline{(\text{Borrow})}(\text{Enable})$	0
(Borrow) $\overline{(\text{Enable})}$	0

QC Modules

The QC Modules summarize the Pass and Borrow conditions of all six groups and will ultimately determine whether or not an End Around Borrow exists.

For the purpose of explanation the following translation of pin 17 = zero is made:

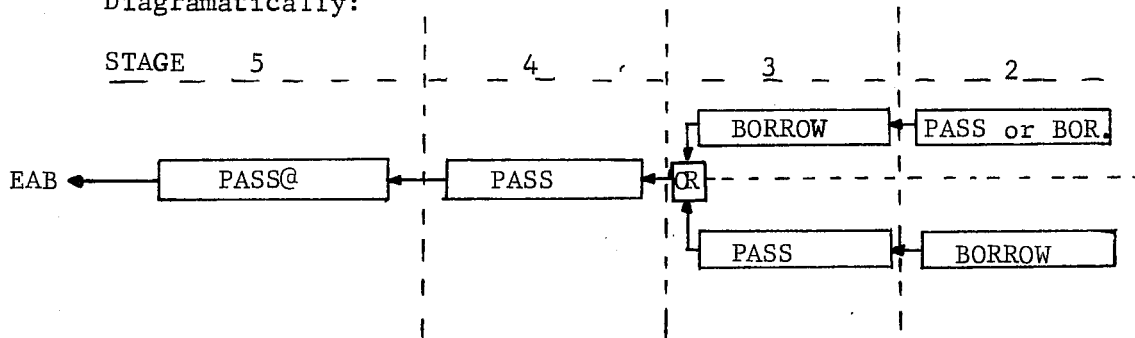
$$\text{Pin 17} = 0 \iff F \cdot B$$

$$F \iff (\text{Gp 5} = \text{Pass})(\text{Gp 4} = \text{Pass})$$

$$B \iff (\text{Gp 3} = \text{Borrow Up}) + (\text{Gp 2} = \text{Borrow Up})(\text{Gp 3} = \text{Pass})$$

$$F \cdot B \iff \left[(\text{Gp 5} = \text{Pass}) \cdot (\text{Gp 4} = \text{Pass}) \cdot \left[(\text{Gp 3} = \text{Borrow Up}) + (\text{Gp 2} = \text{Borrow Up})(\text{Gp 3} = \text{Pass}) \right] \right]$$

Diagrammatically:



In general, if any stage generates a borrow up and all subsequent stages are passes, an End Around Borrow will be generated.

7.7.6 BRANCH TESTS

General Information

The Increment Units test operands for the 04-07 conditional branch instructions and send the test results to the Branch Unit where continuation of the branch sequences are enabled. If the condition is met, the Increment unit sends a "Condition Met" signal to the branch unit and the Jump or Loop sequence is enabled. The absence of "Condition Met" implies that the condition was not met. In this case, the "No Branch" sequence is enabled.

Two branch tests are used to condition the four jump instructions as follows:

Test Used	fm	Condition
Equality	04	$B_i = B_j$
	05	$B_i \neq B_j$
Magnitude	06	$B_i \geq B_j$
	07	$B_i < B_j$

Although both tests use the adder logic, only the magnitude test checks the result of the complete add process. Since B_k (B_j of opcode) is complemented during Branch tests, the adder subtracts B_k from B_j . Hence the result is a difference. The Equality test, on the other hand, checks the 18-bit operands bit by bit.

Equality Test

Figure 7.7-16 is a logic drawing which shows the Branch test circuitry. The equality test simply checks each stage for equality. Since one operand (B_k) is in complement form, equality can be determined by looking for

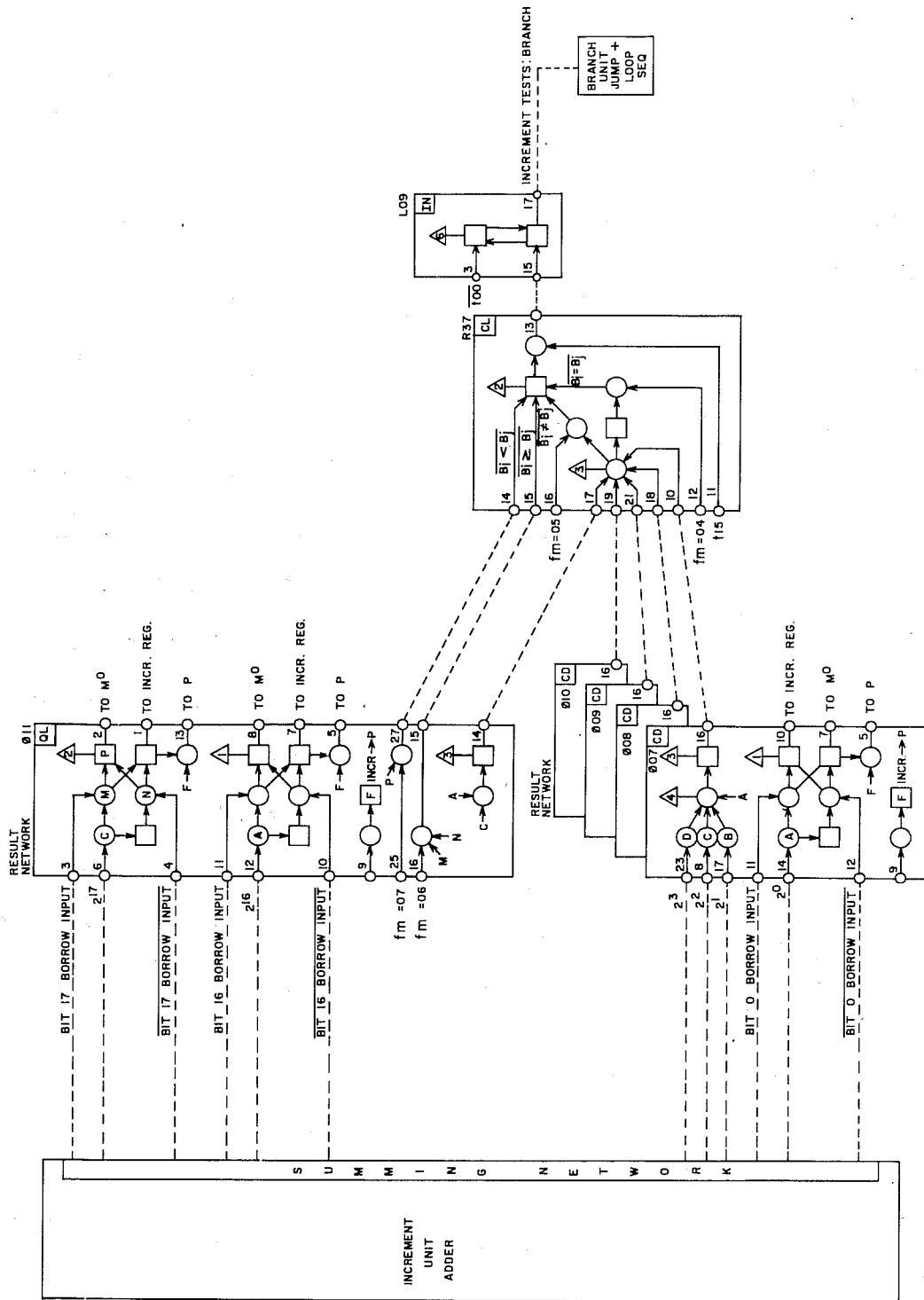


Figure 7.7-16

an "exclusive OR" in each stage. For example,

if feeders equal:

$$\begin{array}{cc} 1 & \text{or} & 0 \\ 0 & & 1 \\ \hline \end{array} \text{(exclusive OR)}$$

if the true operands equal:

$$\begin{array}{cc} 1 & \text{or} & 0 \\ 1 & & 0 \\ \hline \end{array} \text{(equivalence)}$$

Recall that during the adder discussion (Section 7.7-5) it was pointed out that an "exclusive OR" is done with each stage (this was the check for enables). This same signal is used for the Equality test. Pins 14, 17 and 8 on module 007, for instance, state that the content of the feeder flip/flops of stages 0, 1, and 2 (respectively) are equivalent. Since the Bk feeder holds the one's complement of Bk, a zero on pins 0, 1, or 2 indicate equivalence for the given stage. Hence, for a "one" out of TP3, TP4 must have all "ones" in, or stages 0, 1, 2 and 3 must all show equivalence, with respect to true values, or exclusive OR, with respect to the feeder contents. Q08 - Q11 check for equivalence of the remaining stages. If all are equivalent, R37, TP3 will have all "ones" in and a zero out. This condition is ANDed with opcodes 04 and 05. The following combinations yield a "Condition Met":

$$(TP3 = 0 \text{ to Equal})(fm = 04)$$

or

$$(TP3 = 1 \text{ to } \overline{\text{Equal}})(fm = 05)$$

Magnitude Test

The magnitude test simply checks bit 2^{17} of the difference, $B_j - B_k$. If a one (or negative) B_j is considered less than B_k . If a zero (positive) B_j is considered equal to or greater than B_k .

The test is made on module Q11 (Figure 7.7-16). Pin 25 which states that $fm = 07$, is ANDed with term "P" (TP2) which states that bit $2^{17} = 1$ (negative). If the AND conditions are met, a zero appears on pin 27 and the condition is met. Pin 16, which states that $fm = 06$ is ANDed with terms M and N.

$$M \iff \overline{\text{Borrow} + \text{Enable}} \quad (= 0 \text{ sum})$$

$$N \iff \overline{\text{Borrow} + \text{Enable}} \quad (= 0 \text{ sum})$$

By ANDing and simplifying we obtain:

1. $(fm = 06)(M)(N) =$
2. $(fm = 06)(\overline{\text{Borrow} + \text{Enable}})(\overline{\text{Borrow} + \text{Enable}}) =$
3. $(fm = 06)(\overline{\text{Borrow} \cdot \text{Enable}} + \overline{\text{Borrow} \cdot \text{Enable}})$

Formula #3 includes all conditions which yield a zero sum for stage 17, hence the sign is positive and the condition is met.

Module R37, TP2 "OR's" the 4 possible condition met gates:

1. $(fm = 04)(\text{Equivalence})$
2. $(fm = 05)(\overline{\text{Equivalence}})$
3. $(fm = 06)(\text{Result } 2^{17} = 0)$
4. $(fm = 07)(\text{Result } 2^{17} = 1)$

The output of TP2 is gated with a t15 to clear L09, TP6 if the condition was met. The clear side of TP6 feeds pin 17 which in turn sends the "Condition Met" to the branch unit.

SECTION 7.8

BRANCH

Functional Unit

BRANCH FUNCTIONAL UNIT

7.8.1 INTRODUCTION

The function of the Branch Unit is to control the execution of the branch class (fm = 0X) instructions. These instructions may be categorized as follows:

1. Unconditional Branches

- 01 Return Jump to K

- 02 Jump to $B_i + K$

2. Conditional Branches

- 030

- 07 Jump to K if . . .

Handling the unconditional branches is a relatively simple matter of (1) calculating the jump address, (2) placing the new address in the P register, and (3) initiating a memory reference for the new instruction word. An unconditional jump may not be made in the stack, so PK is set to zero, D to 7, and L is set to 7, (indicating IO) and when the new instruction word is read from memory issuance of the instructions begins with parcel zero.

The conditional branches are not quite so straight forward. The branch unit must perform three functions in processing the conditional branches.

1. Determine whether the specified branching condition is met. The operand from an X or a B register is tested by the Long Add or an Increment unit respectively.

BRANCH UNIT BLOCK DIAGRAM

BRANCH F.U.
"INPUT REGISTERS"

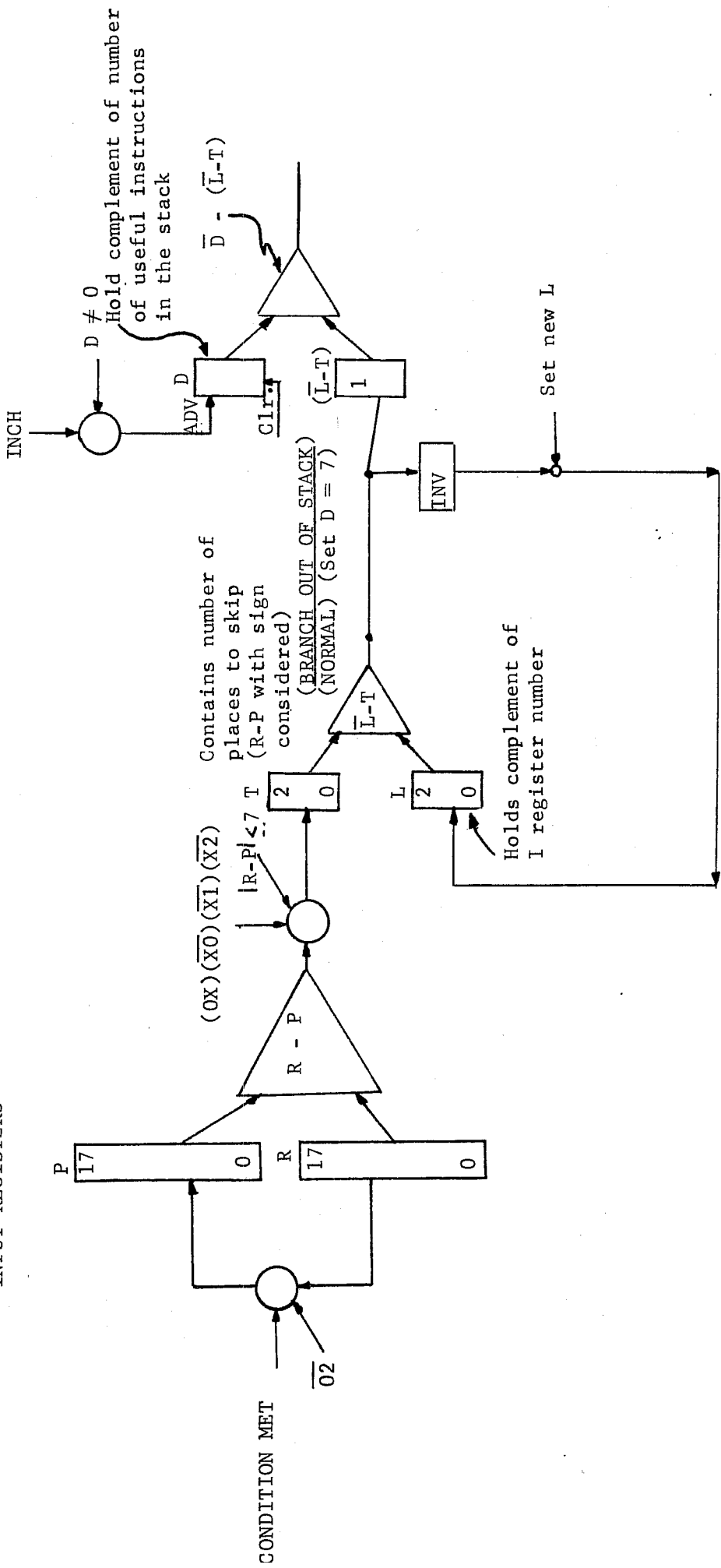


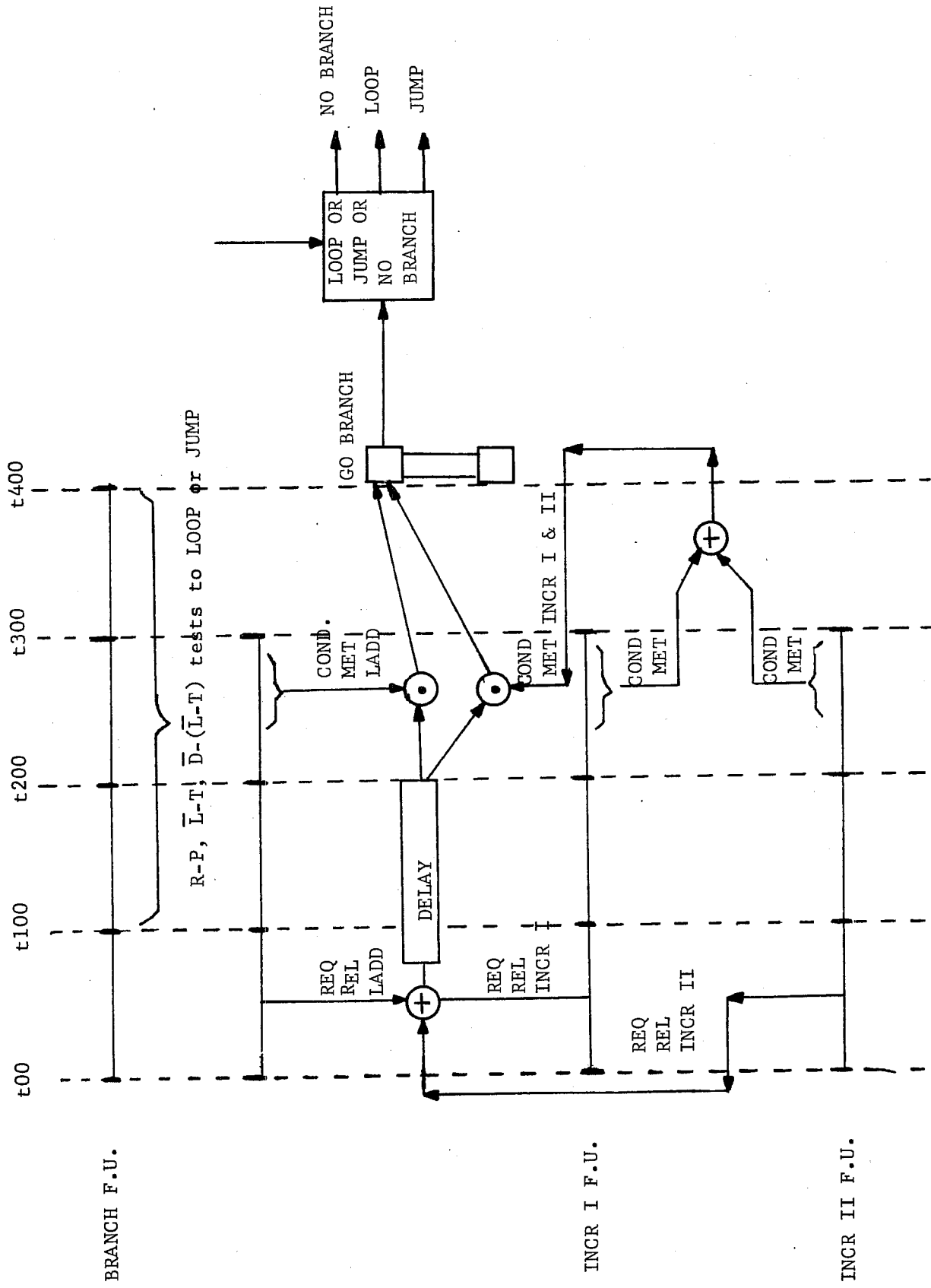
Figure 7.8-1

Results of the tests are sent to the branch unit where they are logically "ANDed" with a translation of the opcode being processed; ultimately, the condition is met, or it is not.

2. If the condition is met a second decision must be made.

Can the branch be made in the stack, or must a memory reference be made for the instruction word? This decision may be divided into three considerations.

- (a) Is the number of places desired to branch within the maximum limit of the physical I registers? A branch in the stack is limited to forward seven registers (from I7 to I0) and backward six registers (I1 to I7). In other words, is the absolute value of the number of places desired to jump less than or equal to seven?
- (b) Is a branch in the stack possible with respect to the I register containing the branch instruction? For example, assume that the branch instruction is located in I4 and a branch backward five places is desired. In this case test 2a will be met since the jump magnitude is less than 7. The maximum jump backward from I4 is three places (to I7); a jump of five is therefore "out of the stack". Thus, although the first test (2a) is met, the second may not be.
- (c) A third and final situation must be taken into consideration, but only in the event that tests one and two are met and a backward branch is desired. Assume, for example, that the branch instruction is in I3 and a branch backward



four places is desired. Both conditions one and two will be met (since we wish to branch from I3 to I7) indicating that the branch is within the physical limits of the stack. Recall that during normal instruction sequencing (RNI) each time a word was sent from memory into IO, the stack was "inched" and the "D register" *was decremented. A D value of zero indicated a full stack; D = 7 indicates an empty stack. Thus, if the branch is backward and within the physical limits of the stack we must determine whether or not enough instructions associated with this particular routine have been loaded into the stack from memory. Returning to the example, if D does not equal zero (full stack) the desired jump to I7 is disabled and a memory reference is started to obtain the instruction word.

Note, that if tests one and two indicate a forward branch in the stack, test three is superfluous since all instructions in the stack after the branch are related to this sequence.

3. The third function of the branch unit is to initiate the new program sequence if a branch is to be made or to continue the old sequence if the branch is not made.

*D holds the complement of the number of instructions in the stack that are within the present subroutine (instruction sequence). It is therefore decremented to increase the value represented.

1. To continue the old sequence (branch condition not met) it is only necessary to generate a "proceed" signal to restart instruction issue. Recall that when stopping issues after the scoreboard issue of the branch instruction, the parcel counter had been properly incremented to select the parcel following the branch instruction. The L register is not changed if a "No Branch" condition exists. Therefore, the generation of the "proceed" will move the instruction following the Branch to U^2 with two U issues and to the scoreboard with the subsequent scoreboard issue. (Refer to Section 7.8.6 for a detailed explanation of the No Branch sequence).

2. To initiate a new program sequence (branch condition met) the two possibilities, Branch In the Stack and Branch Out of the Stack, must be considered.

(a) To Branch In the Stack the stack controls must be modified to select parcel zero of the new instruction word. The parcel counter is therefore set to zero, the L register is loaded with the new value, and the P register is loaded with the jump address (from R). A "proceed" is generated and subsequent issues begin the new "in stack" program. (Refer to Section 7.8.6 for a detailed explanation of the "Loop" Sequence).

(b) In Branching Out of Stack a memory reference (RNI) is required to obtain the next instruction word. The jump address is therefore sent from R to P and the P to M⁰ flip/flop is set. When stunt box priority is granted and the address is accepted, the new instruction word will be sent to the Chassis 5 Input Register. The stack controls are also modified as follows: L and D are both set to 7, and PK is set to 0. Thus, parcel zero of the new word in I⁰ will be the first instruction issued. (Refer to Section 7.8.6 for a detailed explanation of the "Jump" sequence).

7.8.2 INSTRUCTION LIST

The conditional and unconditional Branch instructions are defined in this section. Following each instruction definition is a general explanation of the branch (or no branch) sequence of events. The expressions in parentheses following the instruction name are the ASCENT symbolic codes.

01 Return Jump to K RJ K

Definition:

This instruction stores an unconditional Jump (0400) and the current address plus one (P + 1) in the upper half of address K, then branches to K + 1 for the next instruction. This branch is always out of the stack. A jump to address K at the end of the branch routine returns the program to the original sequence.

Sequence:

The following sequence of events occurs during execution of the return jump:

1. Read Return Jump
2. Stop instruction issue
3. Transfer P (contains P +1) to S register
4. Send R (Jump Address K) to P
5. Send P to M⁰
6. Send S to Memory write distributor and force 0400 into write distributor.
7. Increment P (Jump Address plus 1) and send to M⁰
8. Send M⁰ and tag = 10 (RNI) to Hopper
9. Wait for accept to start issue (proceed)

02 Jump to Bi + K JP Bi + K

Definition:

This instruction branches to the location specified by the sum of register Bi and constant, K. (When i equals zero, the address is K). The branch is always out of the stack.*

Sequence:

An "Out of Stack" (Jump) condition is always forced by the 02 instruction. Thus, the Jump Address is sent to the P register and the P to M⁰ flip/flop is set. Issuance of instructions is

*To perform an unindexed, unconditional jump in the stack, the 04 instruction with i and j = 0 may be used.

resumed when the hopper tag = 10 is accepted.

030	Jump to KYXj = 0	R	Xj	K
031	Jump to KYXj ≠ 0	NZ	Xj	K
032	Jump to KYXj = plus (positive)	PL	Xj	K
033	Jump to KYXj = negative	NG	Xj	K
034	Jump to KYXj is in range	IR	Xj	K
035	Jump to KYXj is out of range	OR	Xj	K
036	Jump to KYXj is definite	DF	Xj	K
037	Jump to KYXj is indefinite	ID	Xj	K

Definition:

These instructions test the 60-bit word in Xj for the condition specified by the i digit. If the condition is met, a jump to K is performed. The tests are performed in the Long Add Unit (See Sections 7.3.2 and 7.3.6 for detailed analysis) and are bound by the following rules:

- (a) The 030 and 031 operations test the 60-bits of Xj for either negative (all ones) or positive (all zeros) zero. All other words are non-zero. The test is valid for fixed or floating print words.
- (b) The 032 and 033 operations examine only the sign (bit 2^{59}) of Xj. If equal to zero, the word is positive; if equal to one, the word is negative. The test is valid for fixed or floating point words.
- (c) The 034 and 035 operations check the upper 12 bits of Xj for either plus or minus infinity. 3777 and 4000 are out of range; all other bit configurations are in

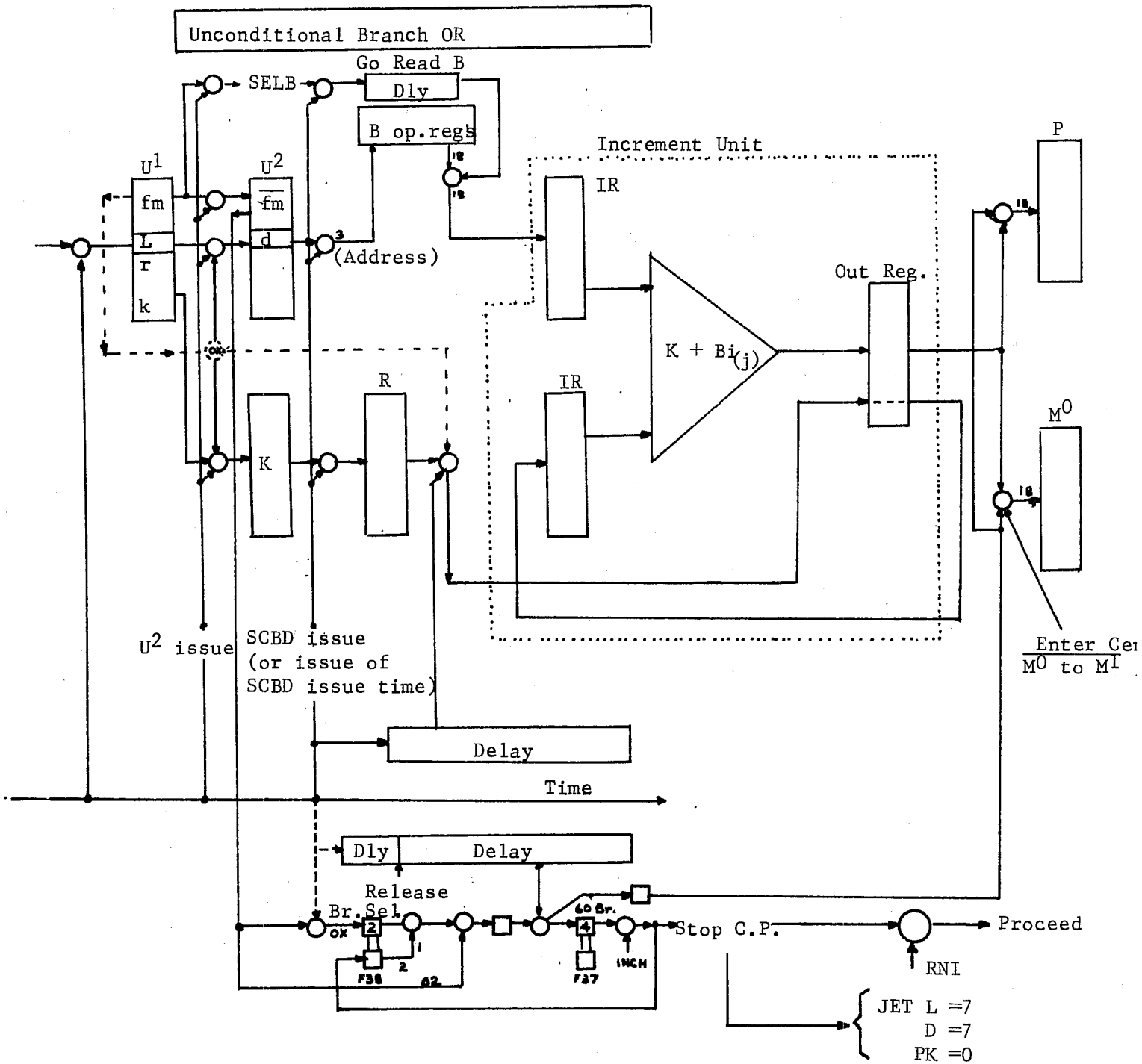
range. The test is valid for both fixed and floating point quantities.

- (d) The 036 and 037 operations test the upper 12 bits of X_j for either plus or minus indefinite forms. 1777 and 6000 are indefinite; all other bit configurations are definite forms. The test is valid only for floating point words.

Sequence:

The 03X instructions cause both the Branch and Long Add units to be initiated at the same time. The Long Add Unit receives the X_j operand from Register Exit control and performs the four tests (zero, sign, infinite, and indefinite) simultaneously. Four signals may result from testing of X_j 1) $X_j \neq 0$, 2) $X_j < 0$, 3) $X_j =$ out of range, or 4) $X_j =$ indefinite. The absence of a signal implies the opposite condition. Thus, eight possibilities exist. The results of the Long Add testing networks are sent to the Branch functional unit, where they are combined with the instruction translation (030, 031, . . . 037) to generate the "condition met" or "condition not met" gates. The Branch unit is informed of the test completion by the "Auxiliary Functional Unit Release" gate which, in this case, is a function of the Long Add Unit's timing chain.

While Long Add is making its tests, the Branch unit is making the In Stack/Out Stack tests. One of two signals, "Loop" (in stack) or "Jump" (out of stack) may result from these tests. They are logically combined with the condition met or not met gates as follows:



(Condition Not Met) • (Loop + Jump) to No Branch

(Condition Met) • (Loop) to Branch In Stack

(Condition Met) • (Jump) to Branch Out of Stack

No Branch:

If the branch condition is not met, the resulting No Branch gate generates a proceed signal which causes the issuance of instructions to resume with the instruction following the branch. Until another branch (OX) instruction is encountered, the normal issue sequence (RNI) takes place.

Loop:

If the Loop and Condition Met gates occur, a Branch in stack will result. In this case, the L register will be loaded with a new value (the "stack address"), the jump address is sent to the P register, and the parcel counter is cleared to zero. A "proceed" is then generated and instruction issue resumes with parcel zero of the new I register.

Jump:

In the event that the Jump and Condition Met gates occur, a memory reference is required to obtain the new instruction word. Thus, 1) the Jump address is sent to P, 2) P is sent to M^0 , 3) M^0 and tag = 10 is sent to M^1 , 4) the D and L registers are set to 7, 5) PK is cleared, and 6) when the tag = 10 is accepted issuance of instructions resumes with parcel zero of IO.

040	Jump to KY $B_i = B_j$	EQ $B_i B_j K$
050	Jump to KY $B_i \neq B_j$	NE $B_i B_j K$
060	Jump to KY $B_i \geq B_j$	GE $B_i B_j K$
070	Jump to KY $B_i < B_j$	LT $B_i B_j K$

Definition:

These instructions test the 18-bit word in B_i against the 18-bit word in B_j (both words are signed quantities) for the condition specified by the opcode. If the condition is met, a jump to K is performed.

The tests are performed in one of the Increment Units (See Section 7.7.6 for detailed analysis). The following rules apply to the tests:

- (a) Positive zero is recognized as unequal to negative zero.
- (b) Positive zero is recognized as greater than negative zero.
- (c) A positive number is greater than a negative number.

Sequence:

The 04 - 07 instructions cause both the Branch and Increment units to be initiated at the same time. The Increment unit receives the two B register operands from Register Exit Control and performs the two tests (equality and threshold) simultaneously. The four possible results ($B_i = B_j$, $B_i \neq B_j$, $B_i \geq B_j$, and $B_i < B_j$) are combined with opcode translations to generate the "condition met" or "condition not met" gates. The Branch Unit is informed of the test completion by the "Auxiliary Functional Unit Release" gate which, in this case, is a function of the Long Add Unit's timing chain.

While the Increment Unit is making its tests, the Branch Unit is making the In Stack/Out Stack tests. The "Loop" or "Jump" gates may result from these tests and are combined with the condition met or not met gates from the Increment Unit.

From this point on, the branch sequence uses the same circuitry as was explained for the O3X branch instructions. Reference is therefore made to the sequence discussion of the O3X instructions for further explanation of the No Branch, Loop, and Jump cases.

7.8.3 TIMING SEQUENCE

Figure 7.8 is a timing chart showing the sequence of events for conditional branches. The auxiliary functional unit used in the chart is an Increment unit. Long Add timing will be similar (both are 300 nanosecond units) and is therefore not shown. The scoreboard issue of the branch instruction is used as the time zero reference.

Refer to the timing chart and Customer Engineering diagrams during the following explanations.

1. Branch Select F.F. is set by $(U^2_{fm} = 0X)$. (SCBD ISSUE)
2. Branch Select F.F. is cleared by the following gate (Q04, pin 16):
 $(fm = 00) + (\overline{GO\ BRANCH}) + (AUX.\ F.U.\ RELEASE \cdot \overline{JUMP} \cdot \overline{LOOP}) + (MC)$
3. The auxiliary functional unit is started at the same time as the Branch Unit.
- 4 - 13. This time series is shown to relate the Increment Unit timing to the Branch Unit. In general, this is the timing of the Increment Unit Branch tests. For detailed explanations of the signals, refer to Section 7.7.3.
14. L21, TP6 is set by the following condition: $(ISSUE) (\overline{ERROR})$. It is cleared by a functional unit reservation code of XXX1. This circuit is a "lockout" which prevents the release from Long Add or the second Increment Unit (which are not associated with the Branch tests) from generating the Auxiliary Functional Unit Release for Branch. Note that of the three flip/flops (L20, 21 and 22 - TP6) only one will remain cleared during a Branch instruction (the one selected by $Q = xxxx$ to perform the test). The other two will be

set by the ISSUE. Thus, only the release from the Functional Unit whose flip/flop is cleared can generate the Auxiliary Functional Unit Release signal.

15. The Auxiliary Functional Unit Release enables the continuation of the Branch sequence. Branch at this point has performed the In Stack/Out Stack test. It now must perform the No Branch, Loop, or Jump sequence.
- 16 - 18. These are stages of the Branch unit timing chain which enable setting the stack controls (if necessary) and proceeding in the proper sequence.
19. The JUMP + LOOP flip/flop is set in one of three ways.
 - 1) Increment Test Condition Met, 2) Long Add Test Condition Met, or 3) Return Jump OR Error Mode AND P to M^0 . It will disable the generation of a Proceed until the Word from CM is sent to IO.
20. The R to P enable is generated for both the Jump and Loop conditions. (Since P must be updated for both the In Stack and Out of Stack branches).
21. If the Jump + Loop flip/flop (R37 - TP4) does not get set (condition not met) the No Branch flip/flop is set. This will enable the sequence which resumes with the instruction following the Branch instruction. L14 - TP6 is the first flip/flop in the sequence.
22. This is a further step in the No Branch sequence (R33, TP5).
23. The Proceed resulting from the No Branch sequence occurs at this time.
24. The "GO" flip/flop is set at this point if the No Branch condition was present.

NOTE: In the event of a Jump, F37, TP4 (flip/flop A) is set. As a result, Stop CP (R32, TP6) and Enable Restart (F37, TP1) are set. In order to "Proceed", the RNI tag (10) must be accepted by CM.

In the case of a Loop, the Branch Unit sets the new value in L and P and clears the Parcel Counter. A Proceed is forced by the Loop signal.

7.8.4 IN STACK/OUT STACK TESTS

The purpose of the in stack/out stack tests is to determine whether or not a conditional branch (O3X or O4-07) can be made in the stack or out of the stack. Ultimately, it is necessary to generate one of two logic signals: "Loop" or "Jump". "Loop" implies an in stack branch. "Jump" implies an out of stack branch. Of course, a third condition exists in case the condition for the branch is not met. This condition is called "no branch". Naturally, in order for a Jump or Loop to be executed, the condition must be met ($\overline{\text{No Branch}}$).

The In Stack/Out Stack Tests are divided into three parts: 1) R - P test, 2) \bar{L} - T test, and 3) \bar{D} - (\bar{L} - T) test. The tests are analyzed separately, but keep in mind that they are inter-related and in the end will indicate the Jump or Loop condition.

R - P TEST:

The R - P test will determine whether the difference between the jump address (K) and the location of the branch instruction (P) is within the limits of the instruction stack (Seven registers forward - I7 to I0, or six registers backward - I1 to I7). This check is made by subtracting the value in P from that in R (R holds the K portion of branch instructions).

Since the network forms R - P, it is logical that if R is greater than or equal to P the result will be positive. A positive difference thus indicates a branch forward. If P is greater than R, a branch backward is implied by the negative result. Also, if the magnitude of the branch is less than or equal to seven, the branch is within the limits of the stack registers.

Four cases may result from the R - P test:

1. R - P positive and ≤ 7 .

Example: R = 010006

P = 010003

Difference: = 000003

The upper bit of the difference equals zero, indicating a positive result, or that $R > P$. Thus, the branch is forward. Bits 3 - 16 of the result equal all zeros. This indicates that the difference was less than or equal to 7. Thus, bits 3 - 17 of the result being all zeros say that the branch is forward seven or less places.

2. R - P positive and > 7 .

Example: R = 010007

P = 007774

Difference: = 000013

Again, the upper bit of the difference equals zero, indicating that R is greater than P. Thus, the branch is forward. Bits 3 - 16 of the result are not equal to all zeros. This indicates that the difference is greater than 7, which means the branch cannot possibly take place in the stack. This is a Jump (out of stack) case.

3. R - P negative and ≤ 7 .

Example: R = 010003

P = 010007

777774

1 ← EAB

777773

The upper bit of the difference is a "one", indicating that P is greater than R. Thus, the branch is backward. Bits 3 - 16 of the result equal all ones. This indicates that the difference is less than or equal to 7. Thus, bits 3 - 17 of the result being all "ones" indicates the branch is backward seven or less places. The difference is the complement of the number of blanched places.

4. R - P negative and > 7.

Example: R = 010005

P = -010016

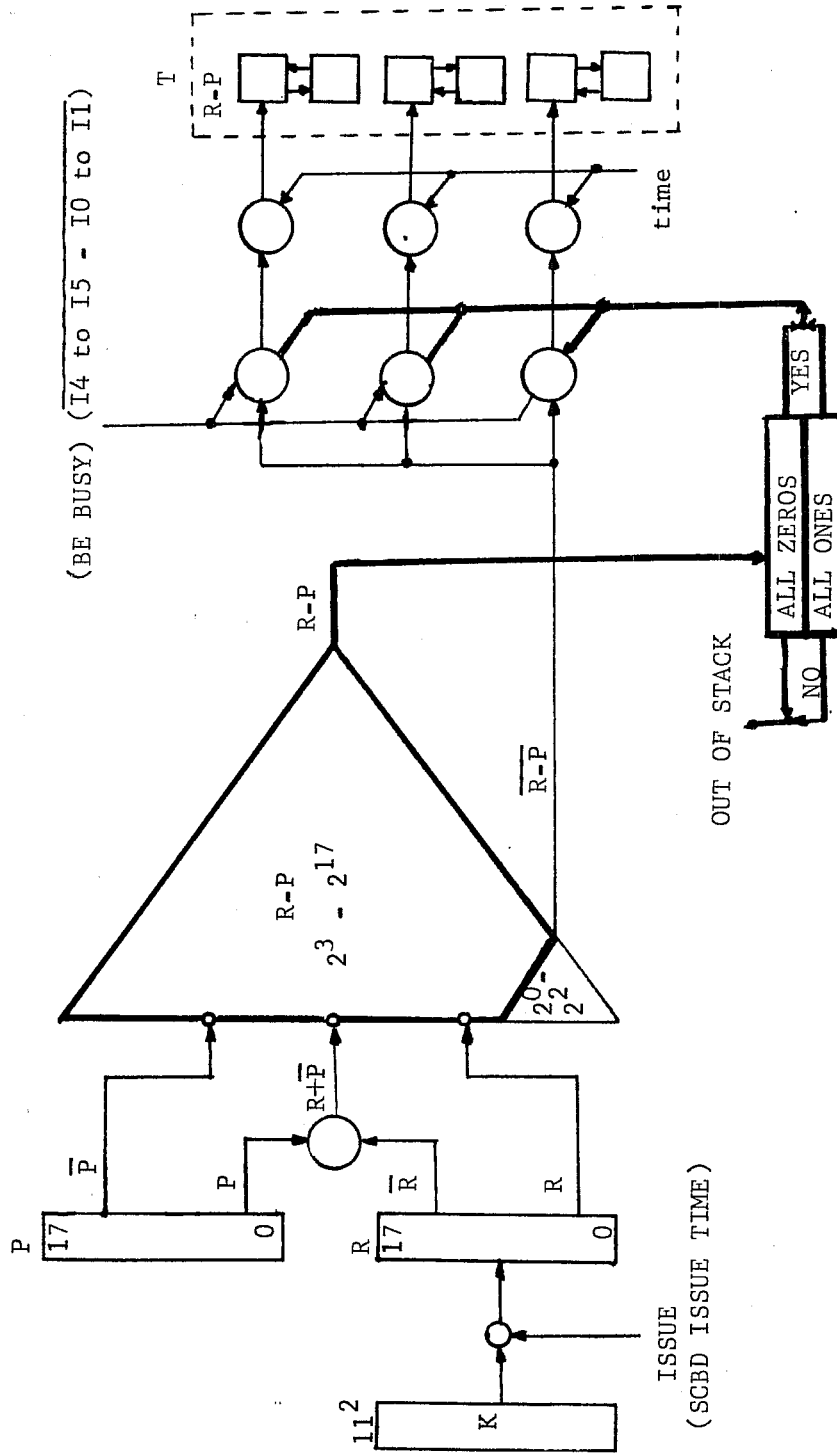
777767

-1 - EAB

777766

The upper bit of the difference (2^{17}) is a one, indicating that P is greater than R. Thus, the branch is backward. Bits 3 - 16 of the result are not all "ones". This indicates that the difference is greater than 7, which means the branch cannot possibly be in the stack (again, a Jump case).

BLOCK DIAGRAM -- R-P to T



Summary

In order for the branch to be in the stack, the absolute value of $(R - P)$ must be less than or equal to 7. This is indicated by the all one or all zero state of bits 3 - 17 of the difference. If these bits are not all ones or all zeros, a Jump (branch out of stack) is forced at this point.

Even though the absolute value of $R - P$ is less than or equal to seven, the "branch out of stack" condition may exist if there are not enough I registers before or after the I register containing the branch instruction. Assume, for example, the branch instruction is in I5 and branch of 6 places forward is desired. A branch forward from I5 is limited to 5 places (from I5 to I0). The branch is therefore out of the stack. A similar situation could exist for a backward branch, where the branch would be out the "top" of the stack. The $\bar{L} - T$ network thus determines whether or not a loop is possible with relation to the position of the Branch instruction. This check is made by the $(\bar{L} - T)$ network which subtracts the result of the $R - P$ test (T) from the complement of the L register. Recall, that the quantity, T, is in one's complement notation. That is, if the result of $R - P$ was negative, T is the complement of the difference; if the result was positive, T is the true value of the difference.

If the result of $R - P$ is negative and less than or equal to 7, recall that bits 3 - 17 of the result are all ones. This condition will force a carry into the $\bar{L} - T$ network. As will be seen, this causes the result of this test to be true for negative values of T (an erroneous answer would result otherwise). If the result of the $R - P$ test is positive and less than or equal to 7, bits 3 - 17 of the result are all ones. In this case, a carry

is not forced into the $\bar{L} - T$ network.

In analyzing the result of the $\bar{L} - T$ network it should be realized that if the result is less than zero (negative), the branch will be out of the stack. This is more obvious with forward branches than with backward.

In the case of forward branch the $\bar{L} - T$ network subtracts the number of places we wish to branch from the I register number where the Branch instruction is located (i.e. \bar{L} yields I register number). Any result of zero means the jump is to I0 (maximum forward loop).

Example: A branch from I4 forward 4 places.



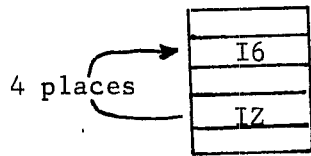
Thus, a difference greater than or equal to zero indicates the branch is forward less than the maximum number of places. (Incidentally, the condition $\bar{L} - T = 0$ is sufficient condition to branch forward in the stack (loop) as we shall see shortly. A negative result ($\bar{L} - T < 0$) then implies the out of stack condition (negative is indicated by an End Around Borrow). In summary of forward branches:

EAB to loop (in stack)

EAB to jump (out of stack)

A branch backward in the stack is not quite so obvious. It would be possible to use the End Around Borrow condition to enable a branch backward in the stack, but if this were the case the true value of T should be added to \bar{L} .

Example: Branch backward 4 places from IZ.



$$L = 5$$

$$\text{branch 4 places (T = 4 - true value)}$$

$$\bar{L} + T = 2 + 4 = 6$$

The result, 6, is positive $\overline{\text{EAB}}$

Note that a branch greater than 5 places backward from IZ would cause overflow (End Around Borrow). $\bar{L} + T = 2 + 6 = 10$. Essentially the same thing is accomplished by subtracting the complement of the desired number of places to branch from \bar{L} . (Since adding can be accomplished by complementing and subtracting).

Using the same example as before:

$$L = 5 \quad (\text{IZ})$$

$$T = 3 \quad (\text{Complement of \# places to branch})$$

$$\bar{L} - T = 2 - \bar{3} = 2 - (-4) = 2 + 4 = 6$$

Note that since we are subtracting, (actual values: $2 - 3$) an EAB will be generated. In the case of negative values of T then, an EAB indicates "in the stack" and NO EAB indicates "out of the stack".

Actually, the $\bar{L} - T$ network operates somewhat differently from the previous example indicated. Recall that if the R - P network generated a negative result less than or equal to 7 (indicated by "all ones") a carry was sent to the $\bar{L} - T$ network. This carry makes the $\bar{L} - T$ adder a two's complement network. It thus forms the quantity expressed as follows:

$$\bar{L} - T - \text{carry} = 1$$

$$(1 = \text{result of } \bar{L} - T \text{ network})$$

If the difference between R and P is positive (forward branch) the carry is not used and the formula becomes simply:

$$\bar{L} - T = 1$$

The output of the $\bar{L} - T$ network (1) will be the true value of the I register into which the branch is desired. If all branch tests are met, the output of the $\bar{L} - T$ network will be complemented and sent to the L register, replacing the old content of L (recall that L holds the complement of the I register number from which instructions are to be issued). The following example should point out the need for two's complement arithmetic:

Assume that the branch instruction is in IZ and a branch backward 5 places (to I7) is desired.

Then: $L = 5, T = 2$

One's complement:

$$\bar{L} - T = 2 - 2 = 0$$

(The result indicates a branch to I0; wrong!)

Two's complement:

$$\bar{L} - T - \text{carry} =$$

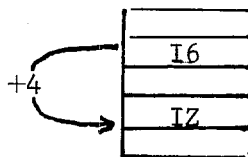
$$2 - 2 - 1 = 7$$

(The result is correct, indicating I7)

To summarize the $\bar{L} - T$ network, the following four cases are presented.

They apply, of course, only if the first consideration (1R - P1 _ is met.

1. Branch Forward In Stack



$T = 4$
all zeros to carry

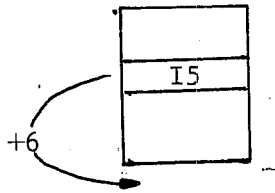
$L = 1$

$$\bar{L} - T = 1$$

$$6 - 4 = 2 \text{ and } \overline{EAB}$$

(all zeros) $\cdot (\overline{EAB})$ to in stack

2. Branch Forward Out of Stack



$T = 6$
all zeros to carry

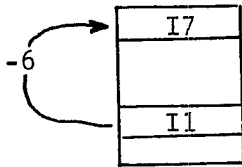
$L = 2$

$$\bar{L} - T = 1$$

$5 - 6 = 7$ and EAB

(all zeros) \cdot (EAB) to out of stack

3. Backward Branch In Stack



$T = 1$
all ones to carry

$L = 6$

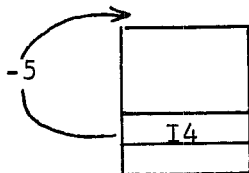
$$\bar{L} - T = 1$$

$1 - 1 - \text{carry} = 7$ and EAB

(all ones) \cdot (EAB) to gate result ($1 = 7$) to the $\bar{D} - (\bar{L} - T)$ network.

Note: Branch is out of stack, but not necessarily in stack. See consideration #3.

4. Backward Branch Out of Stack



$T = 2$
all ones to carry

$L = 3$

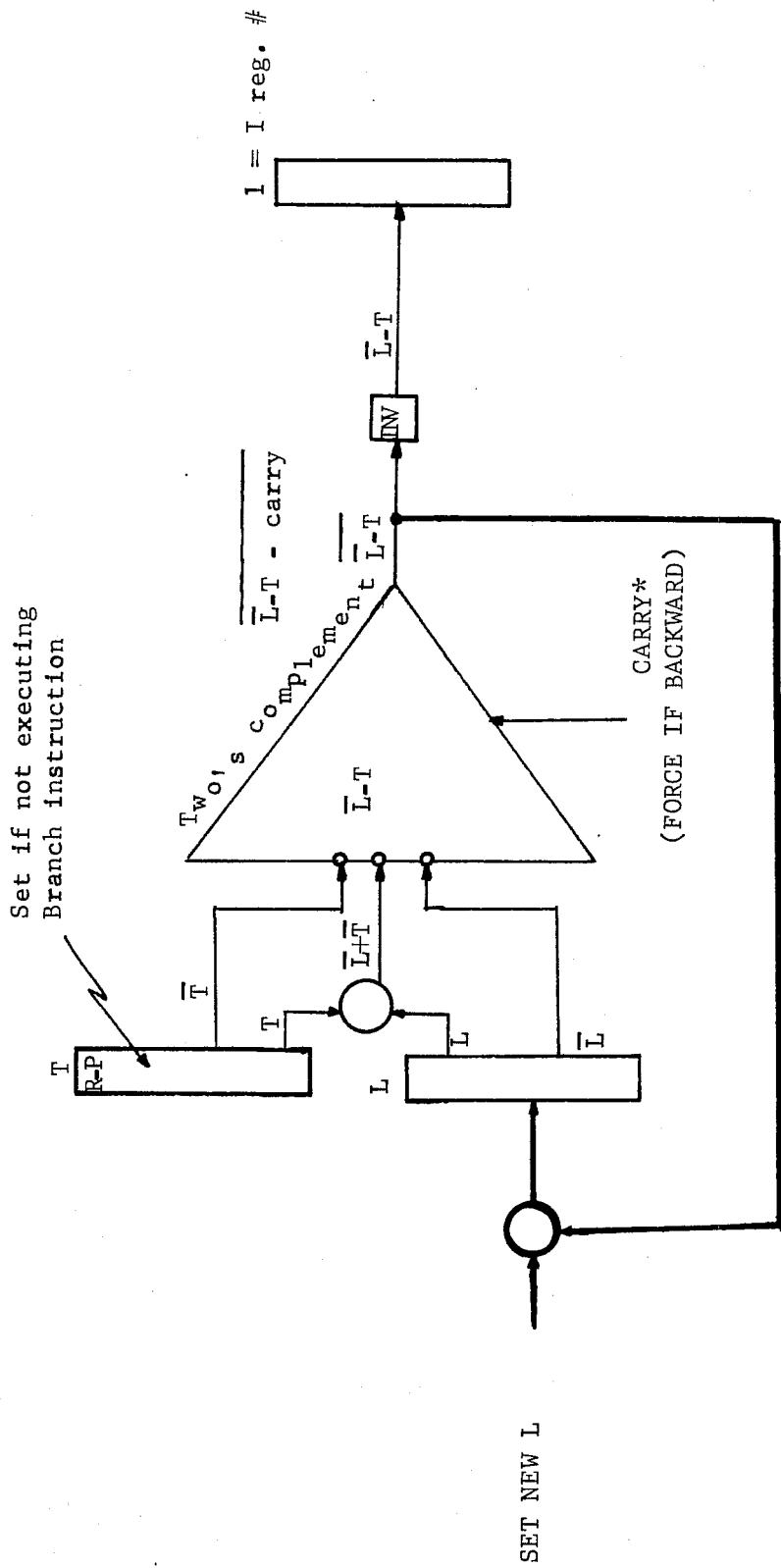
$$\bar{L} - T = 1$$

$4 - 2 - \text{carry} = 1$ and $\overline{\text{EAB}}$

(all ones) \cdot ($\overline{\text{EAB}}$) to don't gate result ($R = 1$) to $\bar{D} - (\bar{L} - T)$ network.

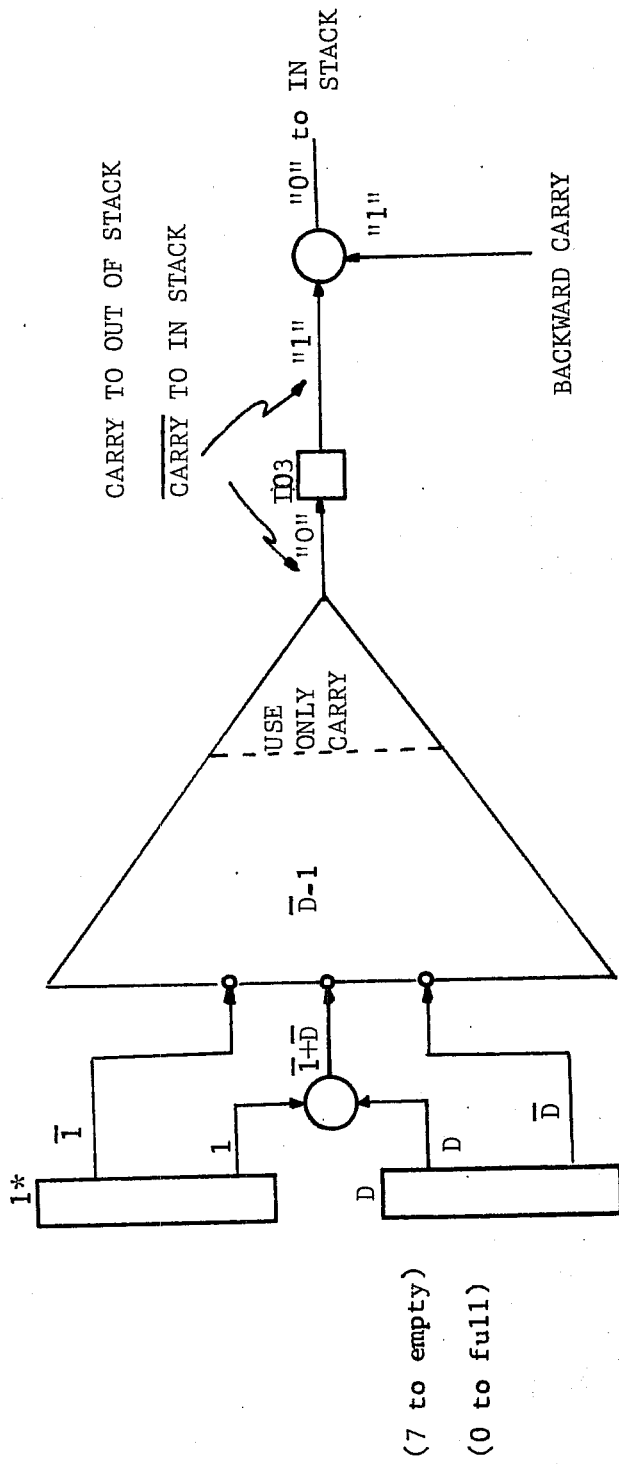
Branch is out of stack.

BLOCK DIAGRAM -- \bar{L} - T



*Makes result true if T was negative

Figure 7.8-3



*1 = result of \bar{L} -T network

Figure 7.8-4

From the previous examples it was seen that the $\bar{D} - (\bar{L} - T)$ test is necessary for only case number 3 of the $\bar{L} - T$ test. With this case it was determined that the branch is backward and within the limit of the physical I registers (i.e. branch is not beyond I7). It is now necessary to determine whether enough valid instructions exist in the I register.

Recall that each time an instruction was brought from memory into IO (RNI) the stack was "Inched" and the D register decremented (D holds the complement of the number of valid instructions). Thus, $D = 7$ indicates an "empty" stack; $D = 0$ a "full" stack.

In analyzing the formula, $\bar{D} - (\bar{L} - T)$, recall that the result of the $\bar{L} - T$ network yields the new I register number. Also, the complement of the D register indicates the number of valid instructions in the stack. From another point of view, \bar{D} indicates the number of the I register which holds the first valid instruction in the stack. For example, D equal to one means there are six (\bar{D}) valid instructions or that the first valid instruction is in I6. Thus, we are actually subtracting the number of the I register to which we wish to branch ($\bar{L} - T$) from the number of the register holding the first valid instruction (\bar{D}). If $\bar{L} - T$ is less than or equal to \bar{D} the difference will be positive (indicated by no End Around Borrow). This means the branch is within the range of valid instructions and the "in stack" gate is enabled. If $\bar{L} - T$ exceeds \bar{D} , we wish to branch to a register number greater than the one holding the first valid instruction; in this case the difference will be negative (indicated by an End Around Borrow). Hence, the range of valid instructions has been exceeded and the "out of stack" gate is generated.

In summary, only the End Around Borrow signal from the $\bar{D} - (\bar{L} - T)$ network is required. The presence of an EAB indicates "out of stack"; the absence of an EAB implies "in stack".

Figure 7.8- is a flow chart of the In Stack/Out Stack tests. It is intended as a logical, concise summary of the decisions made by the test networks as explained in the preceding paragraphs.

7.8.5 UNCONDITIONAL AND RETURN JUMPS

Unconditional Jump (fm = 02)

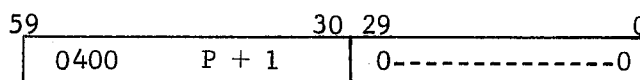
The unconditional jump uses very little of the Branch Units' logic. It, of course, does not use the "Condition Met" circuits nor does it use the "In Stack/Out Stack" tests (since the 02 is always out of the stack).

Nothing prevents the use of the R - P compare network, but whatever the result, it is not sent to the T register. The T register is therefore always set and the out of stack (Jump) condition is always present. (Refer to Figure 7.8- during the following discussion.

The "Go branch" flip/flop (R37, TP4) is set by the translation "fm = 02" (R37, pin 27) and "Release of Auxiliary Functional Unit." The same signal enables P to M^0 which will result in the memory reference at location $K + B_i$. The Go Branch flip/flop is ANDed with the conditions, " $\overline{\text{INCH}}$ " (P10, TP6) and "Out of Stack". This combination of signals sets the Stop CP flip/flop which, in combination with the RNI tag (10) Accepted, will generate the proceed. The setting of Stop CP also sets $L = 7$, $D = 7$, and $PK = 0$. Hence, the next instruction issued will be parcel zero of the Branch Address in memory.

Return Jump (fm = 01)

Although the Return Jump performs a function quite different from the Conditional or Unconditional Jumps, it shares some of the Branch Unit logic. Recall that the 01 instruction stores in location K the following word:



It then transfers program control to location $K + 1$.

To perform these operations, the following events take place (Refer to Figure 7.8-5)

1. Issuance of instructions is stopped as with any OX instruction.
2. The output of the Pincrementer ($P + L$) is sent to the S register with the following gate:

$$(\text{tag} = 60) + (\text{Issue})\overline{(\text{Error})}$$

3. The content of R ($K = \text{Jump Address}$) is sent to the P register with the following gate:

$$(\text{fm} = 01)(P \text{ to } M^0)(\text{Issue})$$

4. P is sent to M^0 with a 50 tag accepted (RJP or EM) and the memory reference (write) is initiated at location K.
5. When the tag = 50 is accepted, the content of the S register ($P+1$) is gated to memory. Also, bit 2^{56} is set and the remaining bits in the write distributor are cleared. Thus, the following word is stored in location K.

(K) =	59	48 47	30 29	0
	0040	P + 1		

6. When the P to M^0 gate is generated address K is sent to M^0 and the Enter Central and Program Address flip/flops are set thus requesting hopper priority 2. The occurrence of P to M^0 gate will clear the RTJ or EM flip/flop (Q04, T1), cause the P register to advance by one, and again set the Program Address and Enter Central flip/flops, thus requesting hopper priority for address $K + 1$.

Since the RTJ or EM flip/flop was cleared, the tag sent to the hopper with address $K + 1$ is a 10 tag (RNI) rather than a 50 tag).

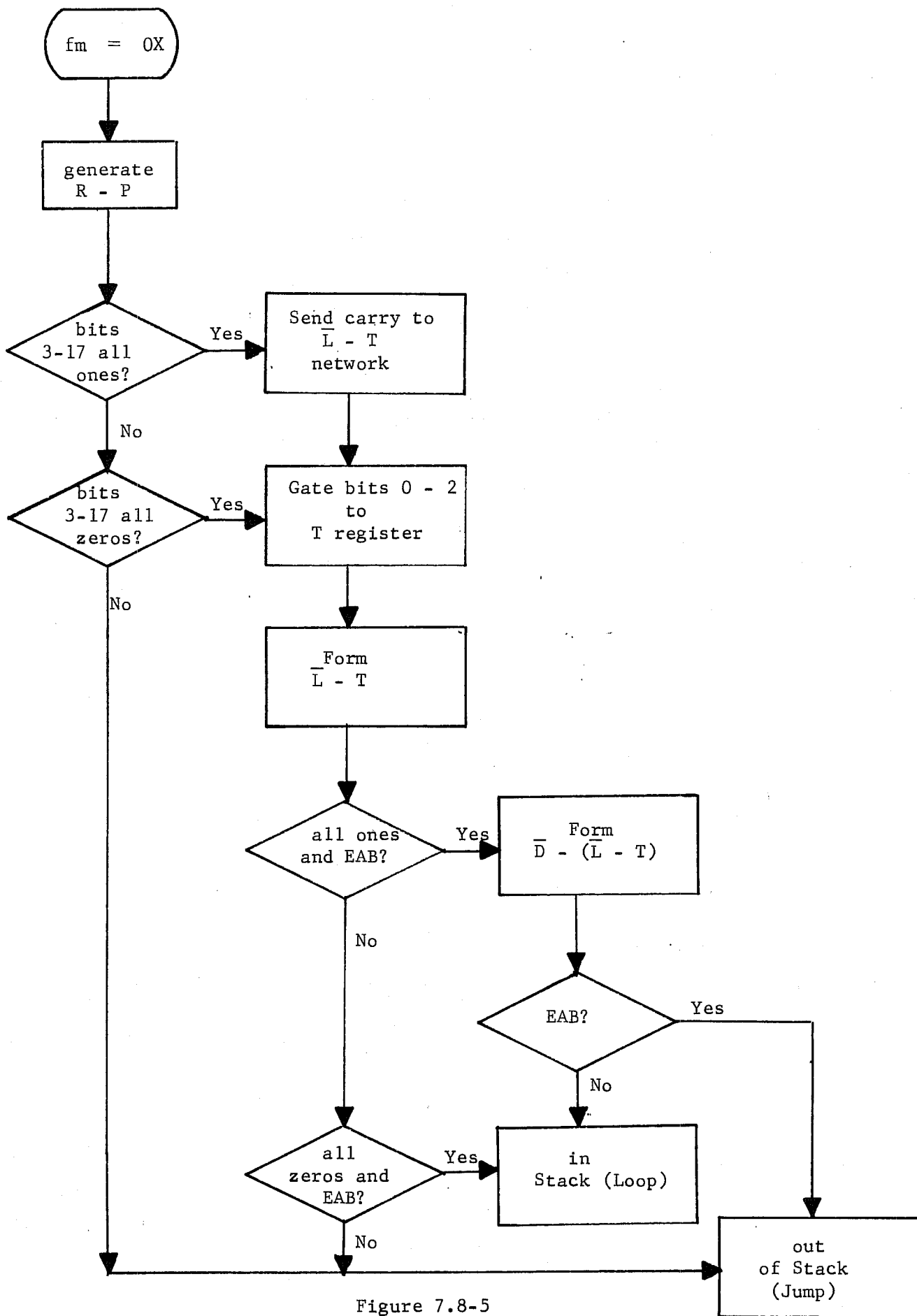


Figure 7.8-5

Thus, when the address is sent through the Read Distributor, it is gated to the Chassis 5 input register as an instruction.

7. The proceed is generated as follows: The "Go Branch" flip/flop (R37) is set by term "X"* the unconditional Jump, the output of the R - P network is not sent to the T register. T therefore remains set and the Out Of Stack gate is always present. Thus, the Jump gate is used to set Stop CP, which again sets L and $D = 7$, $PK = 0$, and sets the Enable Restart flip/flop (F37, TP1). Issue is resumed when the tag = 10 is accepted by memory in the normal fashion.

* Recall, that the Return Jump specifies the jump address as K (not $K + Bi$ as with the unconditional jump). There is no need therefore to start an Increment Unit. Thus, the Auxiliary F.U. Release, which normally is used to set "Go Branch" does not occur. Note, that term "X" (R37) makes no reference to an auxiliary functional unit - the translation is simply $\overline{RTJ + EM}$.

7.8.6 NO BRANCH SEQUENCE

A "No Branch" signal will be generated if a conditional branch instruction is processed, and the branch condition is not met.

The Branch tests are made by an auxiliary functional unit (long Add for the 03X series instructions; Increment I or II for the 04 - 07 instructions) which sends the test results to the Branch Unit. The Branch Unit then determines whether or not the condition has been met by ANDing the test results with the translation of the opcode being processed. This occurs on module R37 (See Figure 7.8-). If the condition is met, the "Go Branch" (Jump or Loop) flip/flop is set (R37, TP4). If the condition is not met, the flip/flop remains cleared.

The cleared state of the "Go Branch" flip/flop disables the generation of "Jump" or "Loop" gates. This is done since P10, TP3 (an AND gate) cannot be made unless "Go Branch" is present. TP3 is needed for both Jump and Loop sequences.

The No Branch sequence is enabled by the cleared state of "Go Branch". If the condition is not met, pin 8 of R37 will be a logical "1". This feeds an AND gate on Q04 (pin 17) whose second input (pin 15) comes from the Auxiliary Functional Unit Release time delay (modules R30 and 31). When both of these inputs are "ones" the output of Q04, pin 21, will be a logical zero. This output clears TP6 on L14 (which is set every time 00). L14, pin 17, enables the setting of R33, TP5. R33, pin 27, is used to disable the \bar{L} - T sequence during No Branch. Pin 25 sets L03, TP6 (via H24, TP4). The Clear side of H24, TP6 (via pin 17) sends a "proceed" to Instruction Go Control which resumes issuing instructions.

Note, that the L and PK registers are not changed by the No Branch sequence.
This means that the instruction following the branch is the next one issued.

7.8.7 LOOP SEQUENCE

Two conditions are necessary to initiate the Loop Sequence. 1) The branch condition must be met. This is indicated by setting the "Go Branch" flip/flop (See Figure 7.8- , R37, TP4). 2) The "In Stack" signal must be present from the Branch Unit's In Stack/Out Stack testing logic.

Setting the "Go Branch" flip/flop disables the No Branch sequence, since the AND gate on Q04 (pins 17 and 15) cannot be made (see Section 7.8.6 - No Branch Sequence). With "Go Branch" set, P10, TP3 will output a "aero" when the Inch flip/flop (P10, TP6) is cleared (term "A"). This zero feeds an "OR" gate which will output a "one" on pin 19 of P10. Pin 19 feeds an AND gate on F33 (pin 26) whose second input (pin 24) says "In Stack". If both of these signals are present, a Loop proceed is sent to Instruction Go Control. Making the AND gate on P10, TP3 places a "one" on one of the inputs to P10, TP2.

The Loop condition must also set the new value of L so that the proper I register can be addressed. This is done by enabling term "G" on G28 (via F33, pin 28) which gates the output of the \bar{L} - T network into the L register for either the Jump or Loop case. (See Figure 7.8-).

Note: As will be seen, the Jump sequence will set $L = 7$ after setting the new value of L. Thus, no problems arise by setting the new L for both Loop and Jump cases. See Section 7.8.8.

The second input (from pin 2) says " $\overline{RT5 + EM}$ ". If these conditions are met, Test Point 2 will cause the transfer of R (contains the jump address, K) to P. Note that this signal will occur in both the Jump and Loop cases. This makes good sense, since the P register should "follow" the

program sequence even when it is executed "in the stack". Note, though, that the Program Address and Enter Central flip/flops will be set only for the Jump case. Therefore, the RNI is not made during Loop.

Finally, the parcel counter must be cleared to insure issuing the first parcel of the Loop program. This is accomplished with the "one" output of F33, pin 23 (Figure 7.8-).

Thus, parcel zero of the I register to which the branch was made is the first instruction to be issued after the branch operation is completed.

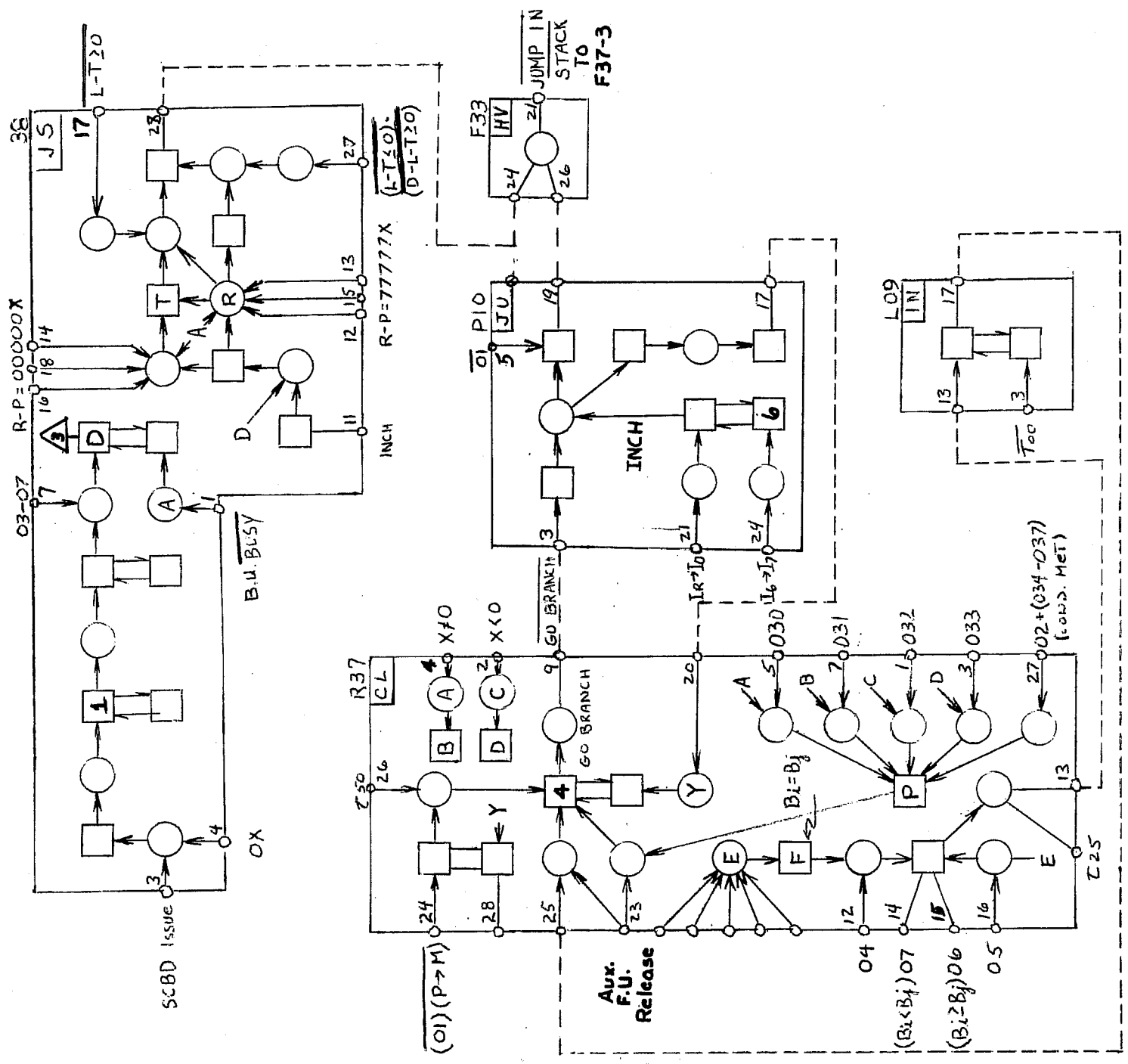
7.8.8 JUMP SEQUENCE

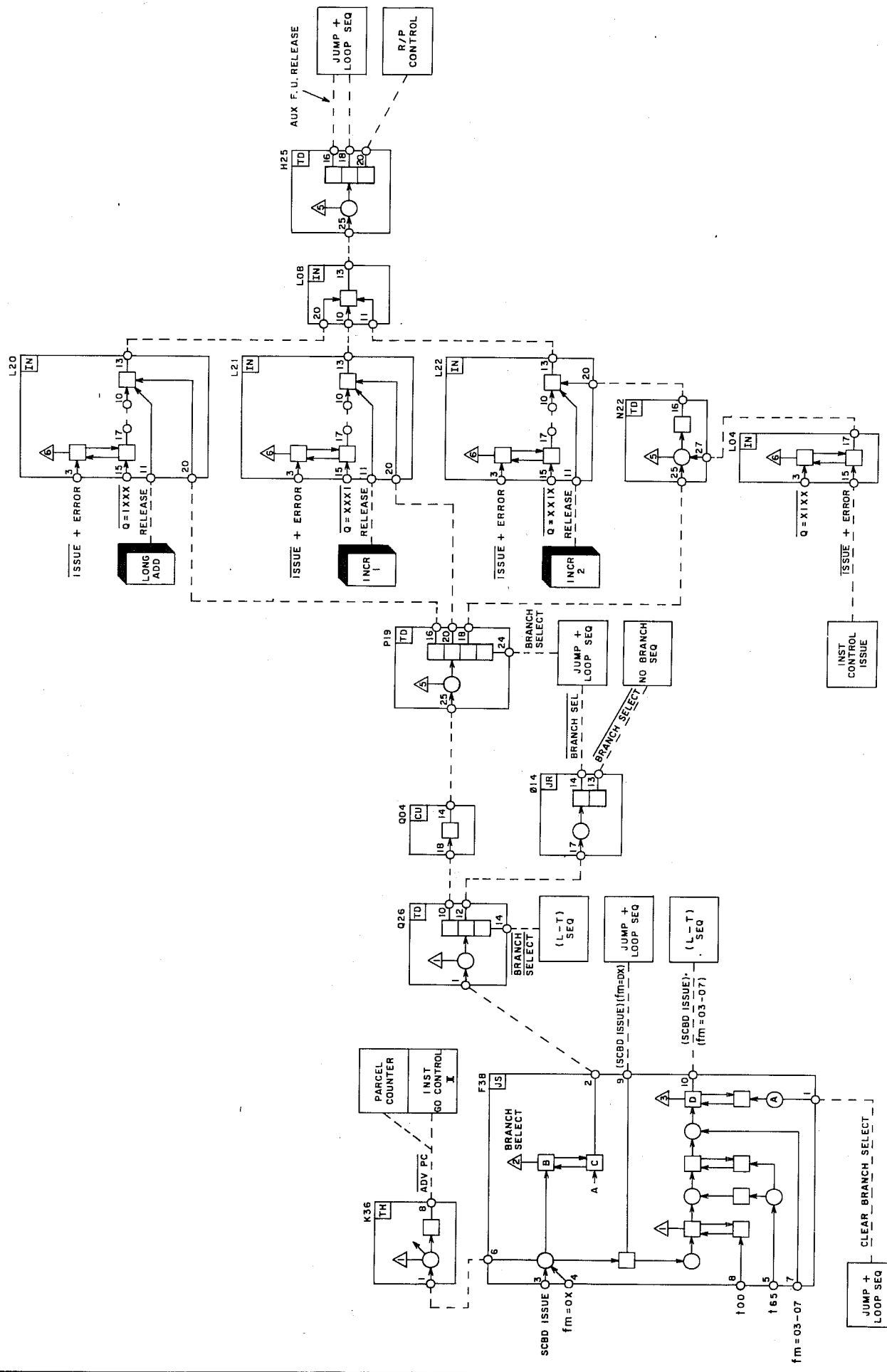
The Jump Sequence is started if a conditional jump condition is met and the branch is out of the stack (as determined by the In Stack/Out Stack tests), or if an Unconditional or a Return Jump is programmed. In any one of the cases, an RNI for the next instruction must be made.

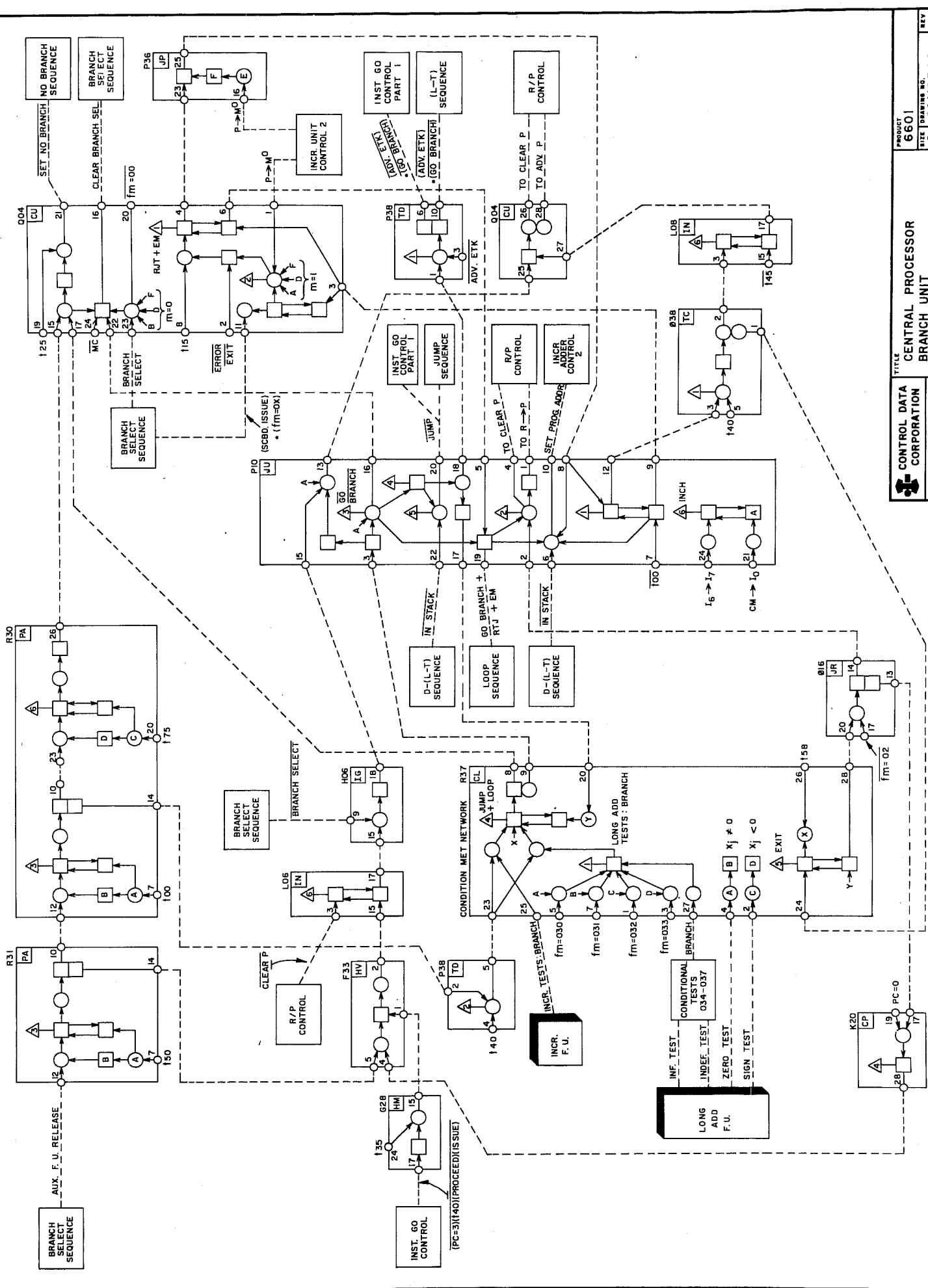
As with the Loop sequence, setting "Go Branch" disables the No Branch sequence since the AND gate on Q04 (pins 17 and 15) cannot be made. The set condition of "Go Branch" allows P10, TP3 to output a zero when the inch flip/flop (TP6) is cleared. TP 3 feeds TP 2 along with the $\overline{\text{RTJ}} + \text{EM}$ gate from 016, pin 14. When both these conditions are present, the R register (contains the Jump address, K) is sent to P. The Program Address and Enter Central flip/flops are set requesting hopper priority. When granted, M^0 will be sent to M^1 along with a tag = 10 (RNI), and the memory reference will be started.


Pin 20 of P10 (translates as $\overline{\text{Go Branch}} \cdot \overline{\text{Out of Stack}}$) feeds pin 21 of F37 (See Figure 7.8-) thus setting TP4. F37, TP4 in turn enables setting the Stop CP flip/flop. This causes L to be set to 7, PK cleared, and the Enable Restart flip/flop to be set. The proceed is thus delayed until the tag = 10 has been accepted.

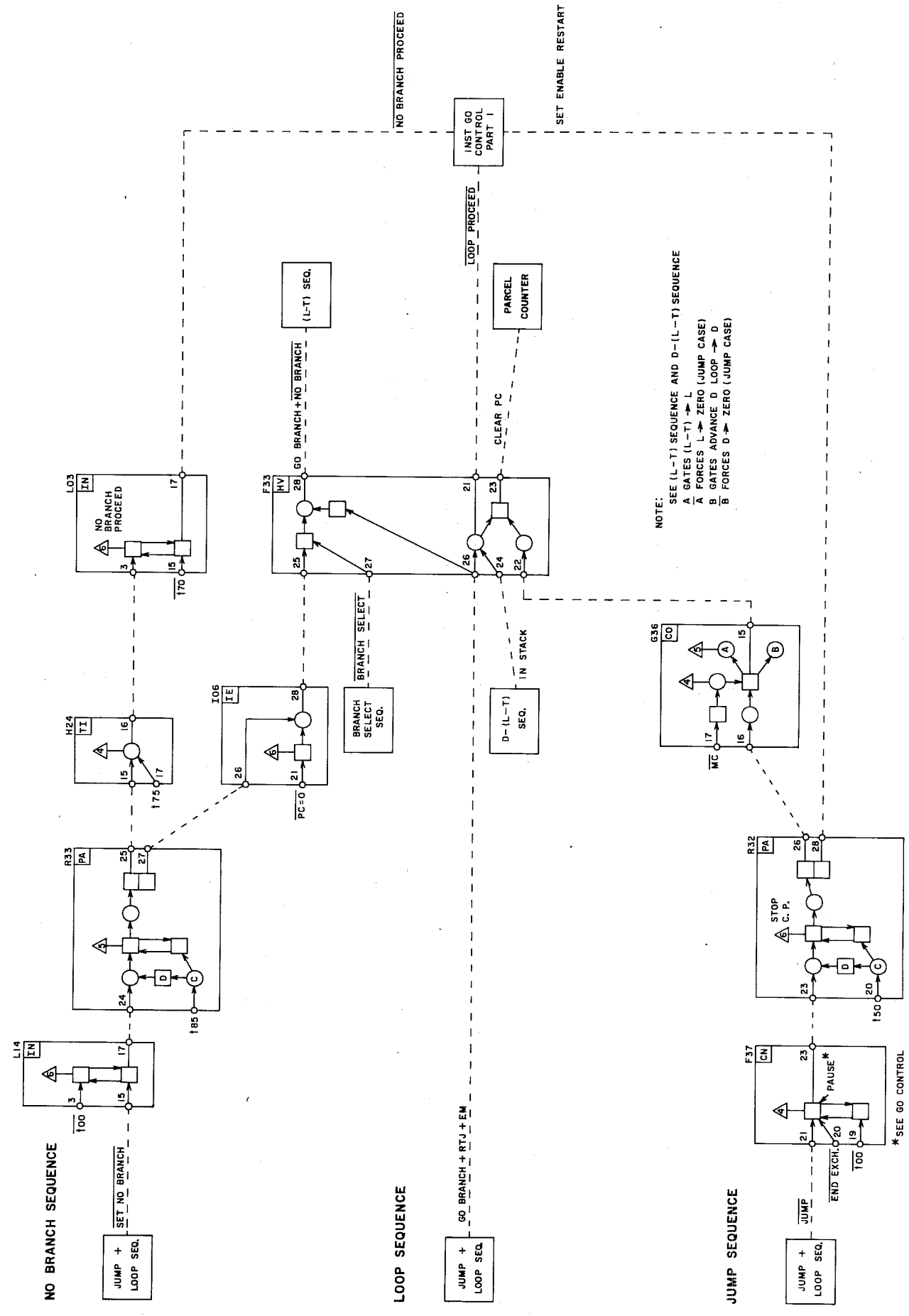
F



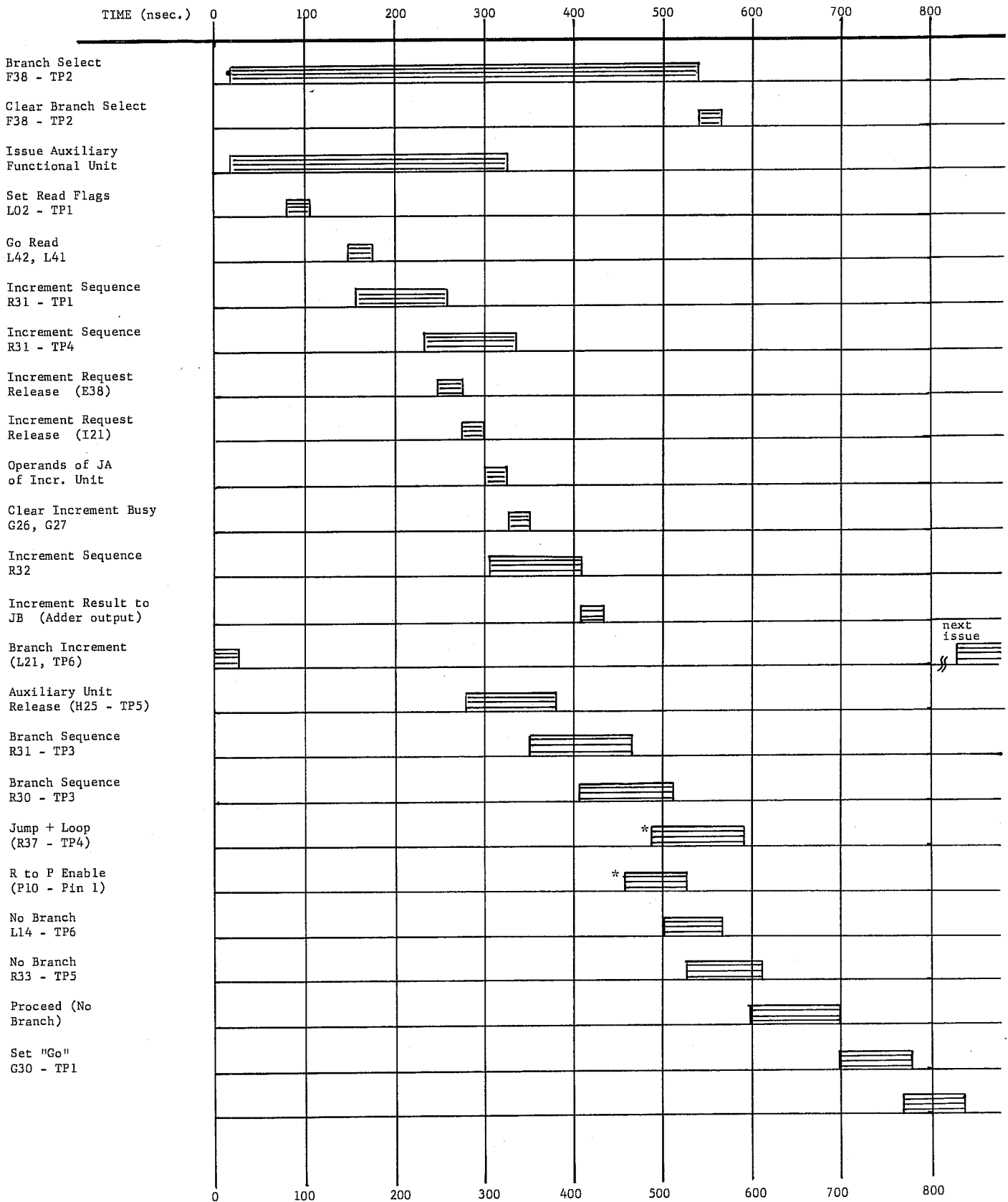




 CONTROL DATA CORPORATION COMPUTER DIVISION	TITLE CENTRAL PROCESSOR BRANCH UNIT JUMP + LOOP SEQUENCE		
	PRODUCT 6601	SIZE DRAWING NO. C 6019300	REV D
COMPUTER DIVISION		SHEET 101	PAGE 11



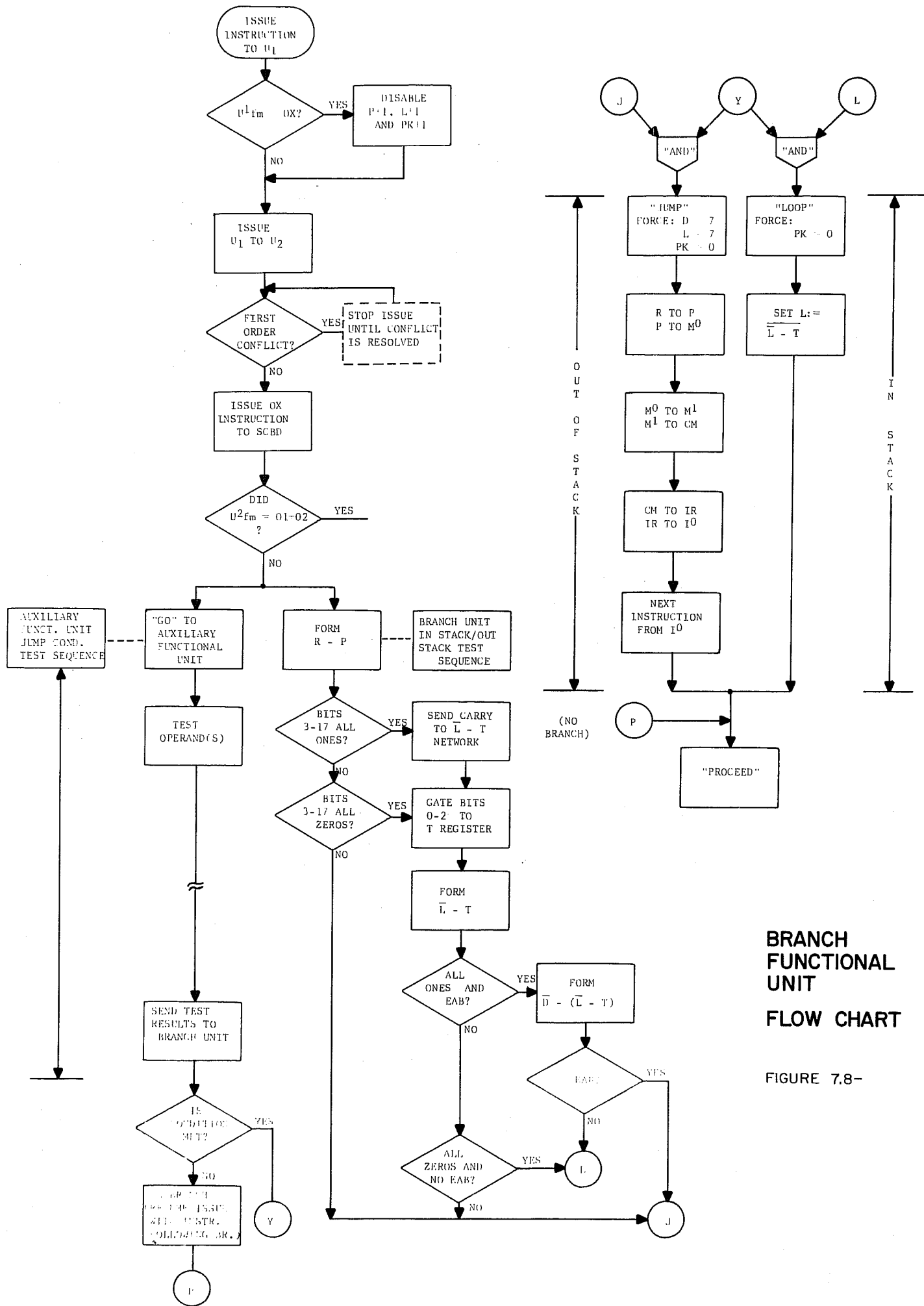
* SEE GO CONTROL



BRANCH UNIT TIMING

* If Branch Test is met

Figure 7.8-



BRANCH FUNCTIONAL UNIT FLOW CHART

FIGURE 7.8-

RETURN JUMP BLOCK DIAGRAM

