CONTROL DATA INSTITUTE

CONTROL DATA CORPORATION

6600

CENTRAL PROCESSOR

Volume II

FUNCTIONAL UNITS

6600 CENTRAL PROCESSOR

Volume II

Functional Units

FOR TRAINING PURPOSES ONLY

This book was compiled and written by members of the instructional staff of

CONTROL DATA INSTITUTE
CONTROL DATA CORPORATION

Publication Number
60239700

History: Former publication number 091466.

FOREWORD

This manual is intended to serve primarily as a reference text for the logic analysis of the 6600 Central Processor Functional Units. During the course of study, the 6600 Central Processor Customer Engineering Diagrams and Wire Tabs should be used as additional reference materials.

The student's thorough knowledge of the following related areas is assumed: 1) 6600 system concept, 2) numbering systems and Boolean Algebra, 3) 6000 Series logic circuits, 4) 6600 Central Processor instruction repertoire and formats and 5) 6600 Central Processor Control and Central Memory logic operation. Information on these subjects can be found in other 6000 Series publications.

The manual is divided into eight sections, each of which deals with one of the eight 6600 Functional Unit types. Two appendices are provided; one deals with 6000 Series floating point arithmetic and the second with the Non-standard operand forms generated by the functional units.

In many cases, the vast amount of logic circuitry utilized prohibits discussion of every detail in the logic. Consequently, great emphasis has been placed on presenting the "concept" of a particular logic circuit. Then, a "representative" analysis of the logic is made. In discussing an adder, for example, an explanation of the purpose of various adder sections is followed by a detailed logic analysis of one adder stage. Since the remaining stages operate similarly, their analysis is left to the student who, by keeping the basic concept in mind, should be able to prove the operation of any stage by use of the wire tabs. "Gaining the concept" cannot be over-stressed when learning a computer system with the magnitude and complexity of the 6600.

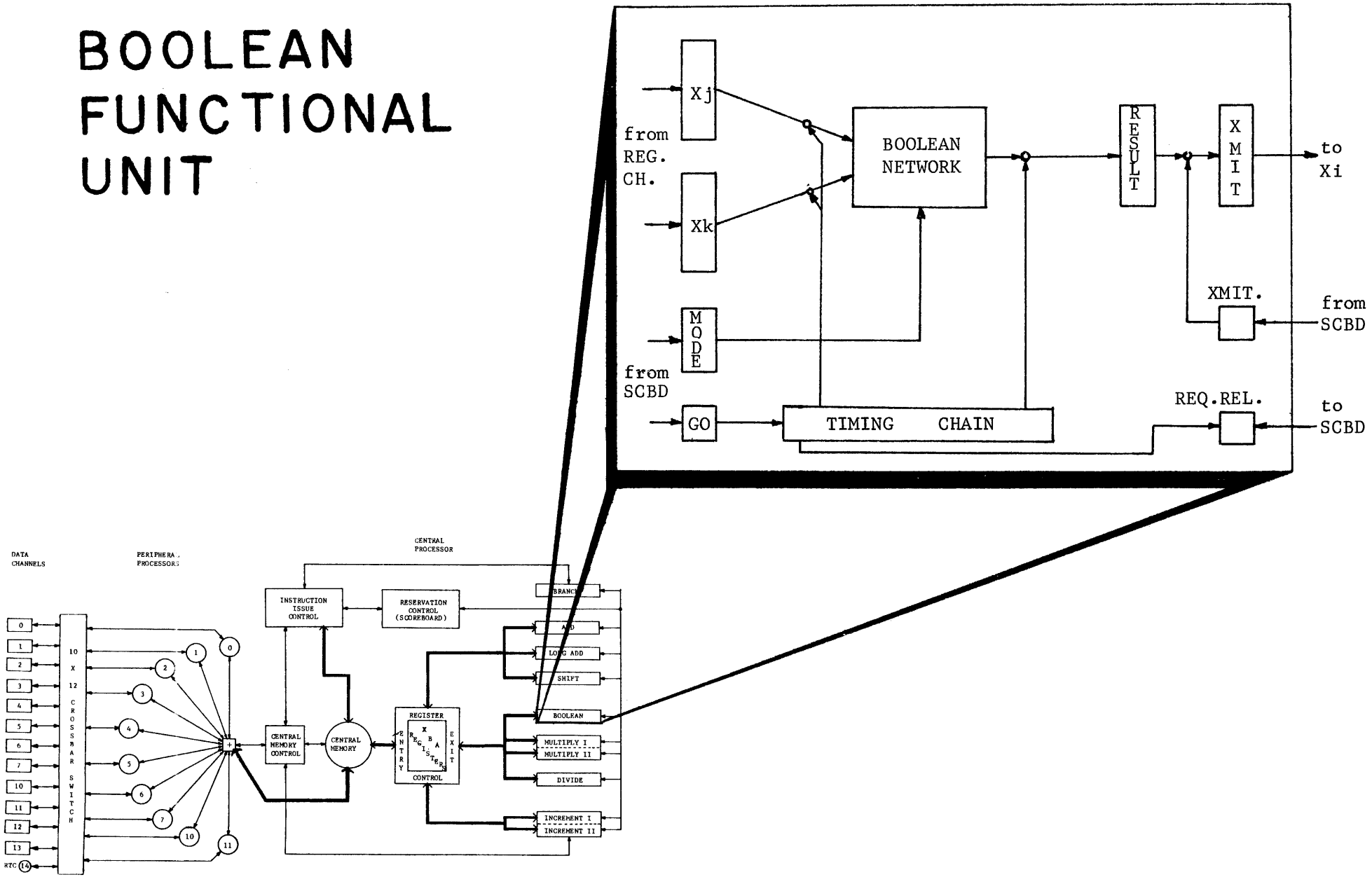
TABLE OF CONTENTS

SECTION 7.1

BOOLEAN

FUNCTIONAL UNIT

BOOLEAN FUNCTIONAL UNIT



BOOLEAN FUNCTIONAL UNIT

7. 1. 1 INTRODUCTION

The Boolean Unit is the least complex of the ten 6600 functional units and therefore is the first one discussed in this chapter. Its concept is relatively simple, yet representative of other functional units. Hence, a thorough comprehension of functional unit principles can be grasped at this point and will make the study of the remaining units less difficult.

The Boolean Unit is a 60-bit, 300 nanosecond unit which performs the logical operations required by instructions 10 through 17 as follows:

10 TRANSFER Xj to Xi

11 LOGICAL PRODUCT of Xj and Xk to Xi

12 LOGICAL SUM of Xj and Xk to Xi

13 LOGICAL DIFFERENCE of Xj and Xk to Xi

14 TRANSFER Xk COMPLEMENT to Xi

15 LOGICAL PRODUCT of Xj and Xk COMPLEMENT to Xi

16 LOGICAL SUM of Xj and Xk COMPLEMENT to Xi

17 LOGICAL DIFFERENCE of Xj and Xk COMPLEMENT to Xi

The Unit is a very basic arrangement of the following components (Refer to the Block Diagram, Figure 7.1-1)

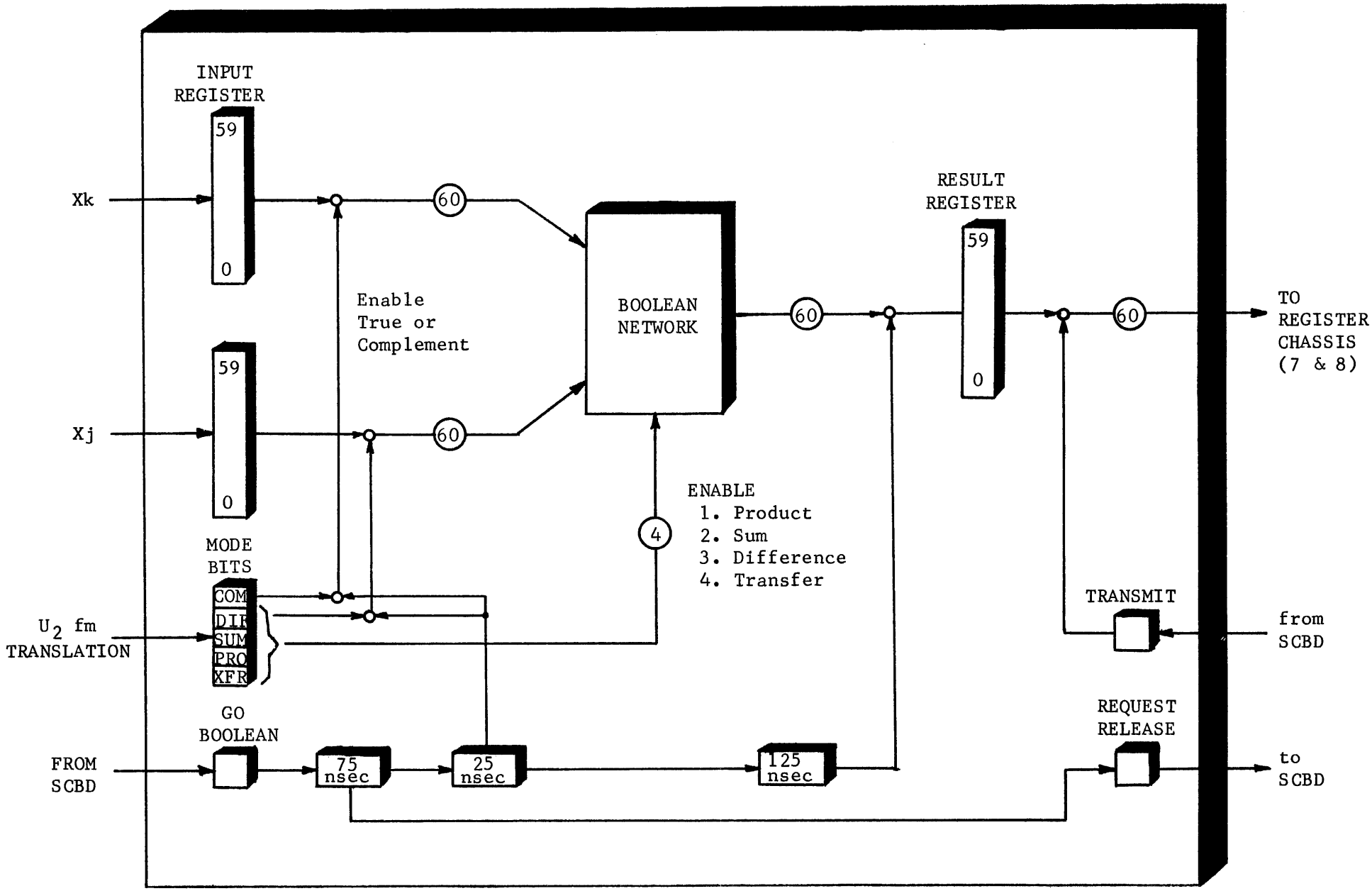
a) two 60-bit input registers

b) a 60-bit result register

c) five mode bits with translators

d) a timing chain

e) a Boolean network



THE BOOLEAN FUNCTIONAL UNIT

Figure 7.1-1

Of these components, the Boolean Network is the unique one. A single GB module contains all the circuitry needed to perform the four logical functions - product, sum, difference and transfer - for three bit positions. Thus, the entire network is embodied in twenty modules (.25% of the total 8000 main frame modules).

The Boolean unit shares data trunk #2 with the Multiply I, Multiply II and Divide units and uses only X registers as source and result registers. It holds last priority on the source operand data trunk and first priority on the result data trunk. It should be noted that bits 0 through 47 of source operands and results travel to and from chassis 2 via chassis 6, while bits 48 through 59 travel directly between chassis 2 and the register chassis. (See Figure 7.1-2)

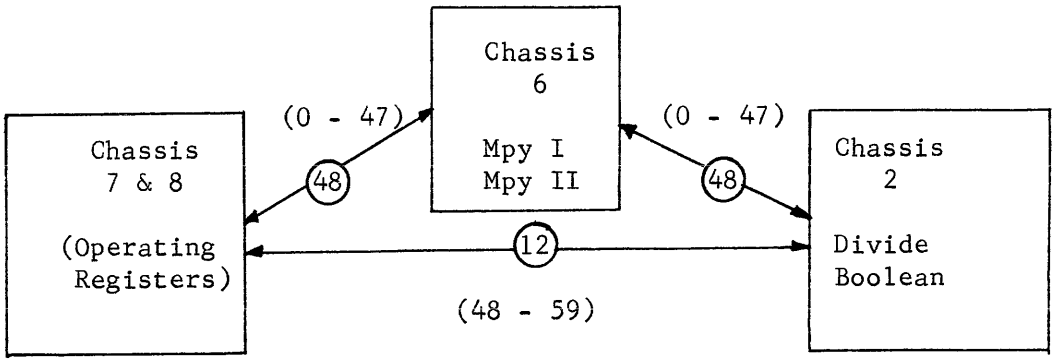
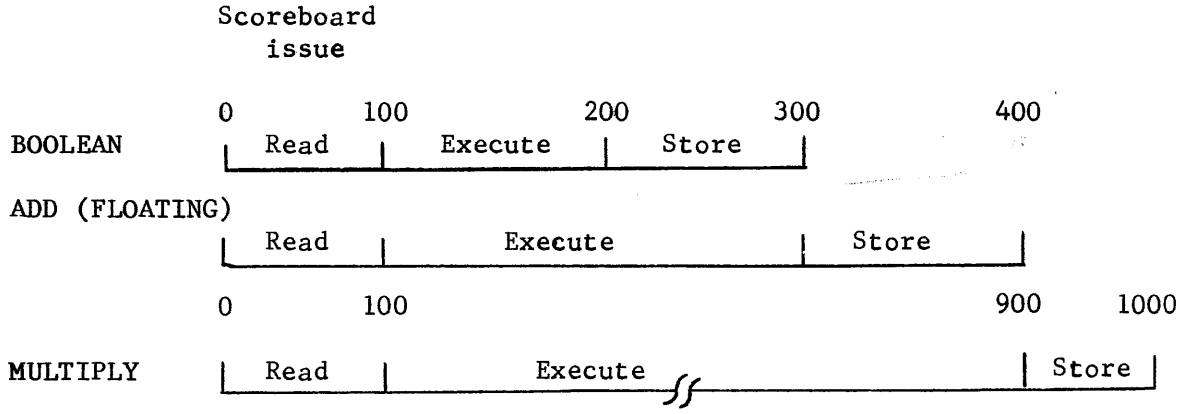


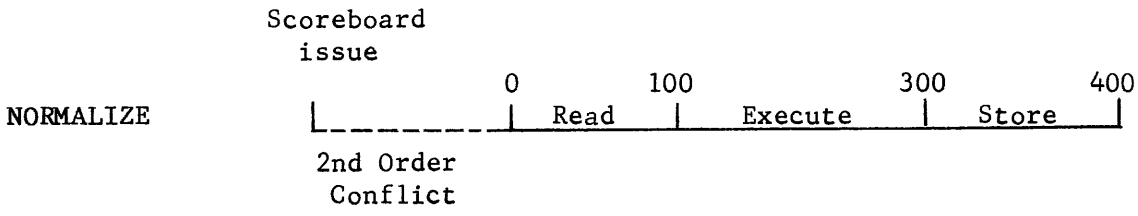
FIGURE 7.1-2

This occurs because the Multiply coefficient logic (bits 0 - 47) is located on chassis #6 while the exponent logic is on chassis #2 along with Boolean and Divide logic. Consequently, bits 48 through 59 need never be sent to chassis #6, and since no timing problems are encountered, the direct route was selected.

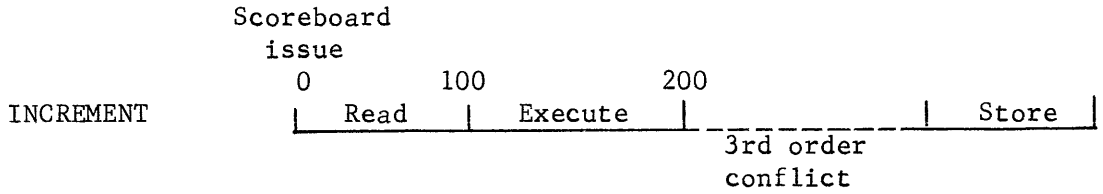
As with all functional units (excepting Branch) the time period of a unit is divided into three categories: 1) Read, 2) Execute, and 3) Store. Read and Store operations each require 100 nanoseconds, while the duration of the Execute portion is a function of the intricacy of the operation required and therefore of the hardware and the number of iterations needed. Study the following examples:



As the examples indicate, the Boolean Unit requires only 100 nanoseconds for the Execute portion while Multiply needs 800. Divide, the slowest functional unit, requires 2.5 microseconds for its Execute portion. If a second-order conflict occurs, there will be a delay between the Scoreboard issue and the Read portion of the unit cycle. Of course, the duration of the delay will depend on the duration of the conflict. When the conflict is resolved (Read flags set) the unit will start. See the following example:



In the event of a third-order conflict, the delay will occur between the Execute and Store portions of the cycle. Again, the duration of the delay depends on when the conflict can be resolved. See the following example.



During the remaining discussion of the Boolean Unit, it is assumed that no second or third order conflicts occur. If one should occur, it simply means that the Read (2nd order) or Store (3rd order) portions of the unit cycle will be delayed until the conflict is resolved.

7.1.2 BOOLEAN INSTRUCTION LIST

In the following definitions, familiarity with Boolean terms and expressions is assumed. If a review is necessary, refer to the Glossary of Logical terms provided at the end of this section. The expressions in parenthesis following the instruction names are the ASCENT symbolic codes.

10 TRANSMIT Xj to Xi (BXi = Xj)

This instruction transfers the content of Xj to Xi. The content of Xk is sent to the Boolean unit, but is ignored. It should be noted that although Xj might not be reserved, a second order conflict with Xk could occur. To prevent this situation, make octal k equal to octal j when coding machine language.* Thus, the conflict can occur only with the source register to be transferred. (i.e., to transmit X5 to X7, use the following code: 10755.)

11 LOGICAL PRODUCT of Xj and Xk to Xi (BXi = Xj · Xk)

This instruction forms in Xi, the Boolean "AND" of Xj and Xk.

*ASCENT (Assembler, Central processor) will automatically make k equal to j.

12 LOGICAL SUM of Xj and Xk to Xi (BXi = Xj + Xk)

This instruction forms in Xi, the Boolean "INCLUSIVE OR" of Xj and Xk.

13 LOGICAL DIFFERENCE of Xj and Xk to Xi (BXi = Xj ⊻ Xk)

This instruction forms in Xi, the Boolean "EXCLUSIVE OR" of Xj and Xk.

14 TRANSMIT Xk COMPLEMENT to Xi (BXi = - Xk)

This instruction transfers the complement of the content of Xk to Xi. The content of Xj is sent to the Boolean Unit, but is ignored. Again, a second order conflict with Xj (the unwanted operand) is possible, so make octal j equal to k when coding. (See the 10 instruction.)

15 LOGICAL PRODUCT of Xj and Xk COMPLEMENT to Xi (BXi = -Xk · Xj)

This instruction forms in Xi, the Boolean "AND" of Xj and the complement of Xk. In terms of the true source operands (Xj and Xk NOT complemented) this instruction forms in Xi the "SELECTIVE CLEAR" of Xj conditioned by Xk.

16 LOGICAL SUM of Xj and Xk COMPLEMENT to Xi (BXi = -Xk + Xj)

This instruction forms in Xi, the Boolean "INCLUSIVE OR" of Xj and the complement of Xk. In terms of the true source operands this instruction forms in Xi the "IMPLICATION" of Xj by Xk.

17 LOGICAL DIFFERENCE of Xj and Xk COMPLEMENT to Xi (BXi = -Xk ⊻ Xj)

This instruction forms in Xi, the Boolean "EXCLUSIVE OR" of Xj and the complement of Xk. In terms of the true source operands this instruction forms in Xi the "EQUIVALENCE" of Xj and Xk.

Note that the instructions can be arranged in a 2 x 4 matrix as follows:

	Transmit	L. Prod.	L. Sum	L. Diff.
Xj and Xk	10	11	12	13
Xj and -Xk	14	15	16	17

FIGURE 7.1-3

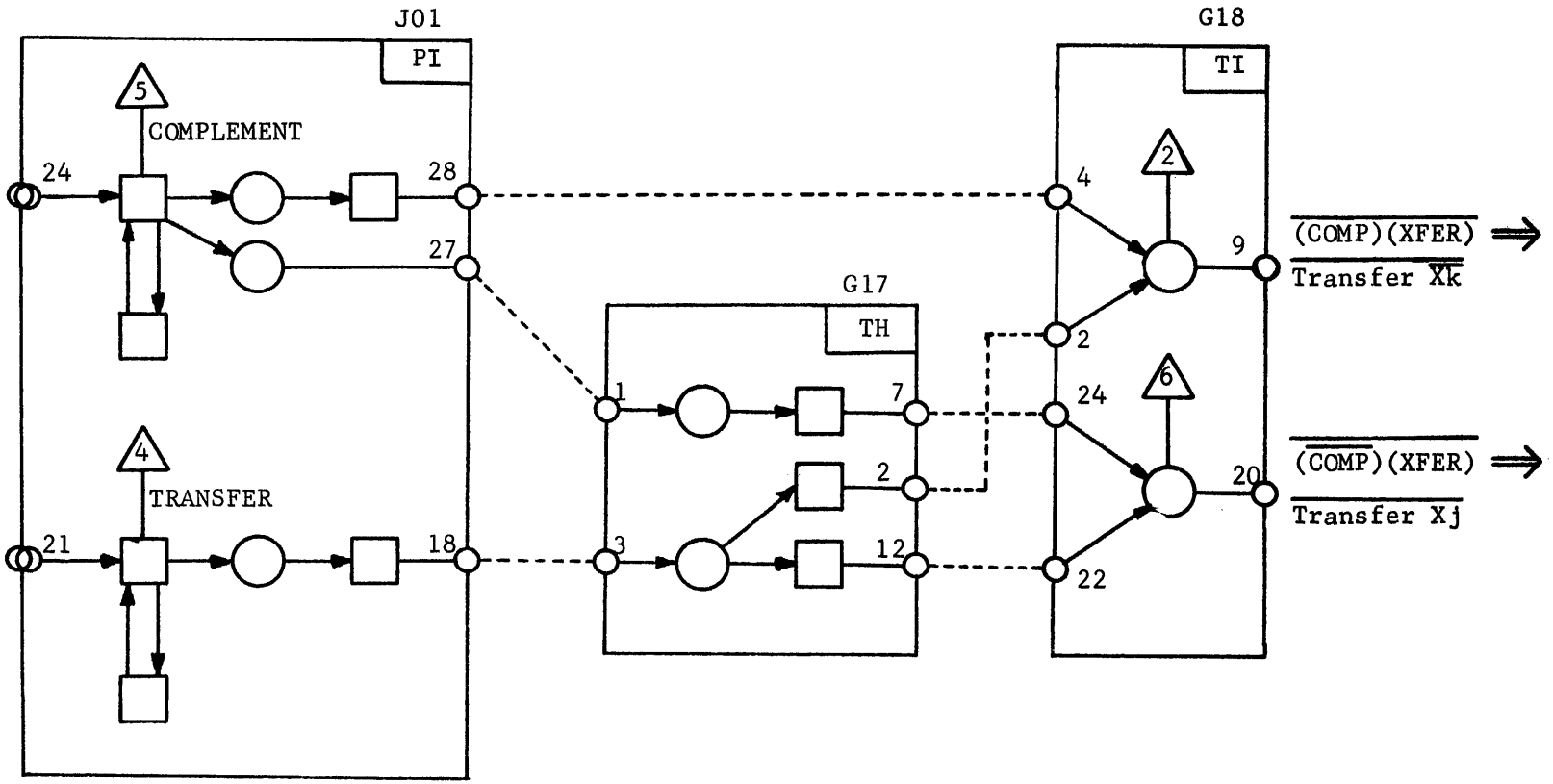
7.1.3 MODE BITS

Now that the instruction requirements are known, it is appropriate to see how the logic determines exactly which functions to perform. This is the job of the five mode bits, which were sent to the Boolean Unit upon issuing the instruction to the Scoreboard. (U register translators select the Functional Unit and enable the required mode bits to that unit.)

By referring to the 2 x 4 matrix (Figure 7.1-3) once again, differentiating the eight Boolean instructions becomes quite easy. Four logical operations can be performed: 1) Transmit, 2) Logical Product, 3) Logical Sum and 4) Logical Difference. Note also that the true value of the Xj operand is always used, while the true value of Xk is used only by instruction 10 through 13. Instructions 14 through 17 use the complement of Xk. In summarizing these conditions, the following list of mode bits and the fm translations selecting each mode bit can be derived:

MODE BITS	fm (INSTRUCTION)
Transfer	10, 14
Product	11, 15
Sum	12, 16
Difference	13, 17
Complement	14, 15, 16, 17

For any Boolean instruction, only one of the first four mode bits is sent to indicate which of the four logical functions to perform. The Complement



TYPICAL MODE BIT TRANSLATION

FIGURE 7.1-4

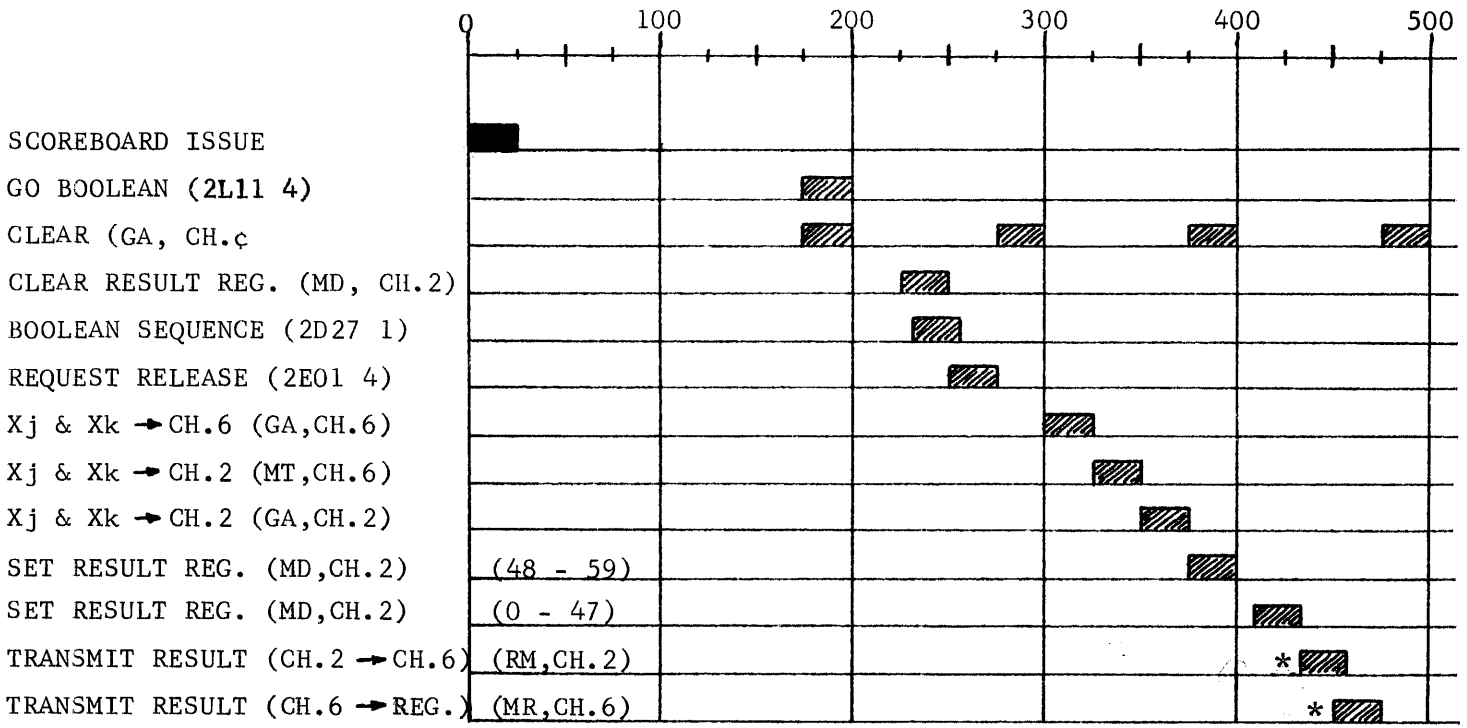
mode bit is used to determine whether to use the true or complemented value of Xk. Thus, a basic "AND" of each of the first four mode bits with the Complement mode bit will select one of the eight distinct operations. For example, if both transmit and complement mode bits are received, the translators will enable a transfer of Xk complemented to the result register. If only Transmit was received, the implication is to gate Xj to the result register (Refer to Figure 1.7-4). The same method applies to the Product, Sum, and Difference functions.

For complete illustration of the mode bit logic, reference is made to the 6601/4 Customer Engineering Diagrams, sheet 105, where the receivers, translators, and fan-outs can be seen. Note how the mode bits are "ORed" together on the TI modules and fanned out on the GAs. From the fan-outs, the enables gate the Boolean network and input registers. Before analyzing their effect on the network, timing of the Boolean Unit will be discussed.

7.1.4 TIMING SEQUENCE

The timing sequence of the Boolean Unit is quite straightforward, but two considerations should be kept in mind. First, the Boolean Network itself is a static network (no timing gates) and, consequently, a logical result of the inputs to the network will always be seen at the output. (The result, of course, depends upon the gates enabled and disabled by the mode bits.) Second, recall that bits 48 - 59 of the operands arrive before bits 0 - 47. (As we discussed earlier, 48 - 59 take the direct route; 0 - 47 arrive via chassis 6) It will also be seen that bits 0 - 47 of the result are transmitted before 48 - 59 which again have the shorter path. This occurs so that all bits arrive at Register Entry Control at about the same time.

BOOLEAN FUNCTIONAL UNIT - TIMING CHART



* Earliest possible time - No Result Register Conflict.

FIGURE 7.1-5

EXPLANATION OF THE TIMING CHART:

t000 - time of the SCOREBOARD ISSUE

t175 - The Boolean Unit receives "Go Boolean" at L11, TP4 (Prints, Sheet 105)

t225 - Clear Boolean Result Reg. "Clear" is fanned out from N11 (Sheet 105) and Clears all the MD modules (Sheet 103)

t230 - Set the second Flip-flop in the Boolean timing chain (Sheet 105, D27, TP1)

t250 - Send "Request Release" to the Scoreboard (Sheet 105, E01, TP2)

t300 - 1) Receive bits 48 - 59 of source operands on chassis 2 (Sheet 103, GA's) and 0-47 on chassis 6 (GA's)

2) Set third Flip-flop in the Boolean timing chain (Sheet 105, D27, TP4)

3) Enable Xj and Xk (true or false) to the GB modules (Sheet 103)

t325 - Transmit bits 0 - 47 from chassis 6 (MT modules) to chassis 2.

t350 - Receive bits 0 - 47 of source operands on chassis 2 (Sheet 103, GA modules)

t375 - Set Result Register for bits 48 - 59 (Ø10 on Sheet 105 fans out to MD modules on Sheet 103)

NOTE: Since the Boolean Network is a static network, the time required to generate a result is a function of the longest inverter path from the Input Register to the input pin of the Output Register. The longest path is seven inverters (about 35 nsec.). Note from the timing chart that approximately 75 nsec. are allowed from receipt of the operands until the result register is set.

t410 - Set Result Register for bits 0 - 47 (Sheet 103, MD modules)

NOTE: It must be assumed at this point that the "Request Release" send at t250 encountered no third order conflicts in the scoreboard. Accordingly, a "Transmit" signal will be received about t375 (Sheet 105, L11, TP6)

t430 - Transmit bits 0 - 47 from chassis 2 (Ø11 on Sheet 105 fans out to MD modules on Sheet 103)

t455 - 1) Transmit bits 0 - 47 from chassis 6 to Register chassis

2) Transmit bits 48 - 49 from chassis 2 (Sheet 103, RM modules) to Register chassis.

It is assumed, for the purpose of simplicity, that no second order conflicts or trunk priority conflicts occur after issuing the Boolean instruction to the scoreboard. Accordingly, the Boolean Unit will receive a "Go Boolean" pulse about 145 nanoseconds after scoreboard issue. This pulse starts the timing chain which causes the events shown on the Timing chart (Figure 7.1-5) and page page 13 to occur. (Module, pin number, and test point references are made to facilitate the following of signals through the logic in the Customer Engineering Diagrams.)

It is interesting to note that due to the short filter time of the Boolean network the "Request Release" is transmitted prior to the receipt of the source operands!

7.1.5 THE BOOLEAN NETWORK

During this discussion, refer to the chart on page 7.1.14 and the Boolean Network (7.1.15). Module and pin numbers referenced in this explanation pertain to the network for bit 2^0.

As was discussed earlier the Boolean network, per se, is composed of twenty GB modules, each containing the logic circuitry for three bit positions. The input registers (fourteen GA modules and one KU module for each operand) feed the Boolean network. The output of the GB modules is stored in the result register (MD modules) and transmitted from the RM modules upon receipt of the "Transmit Boolean" signal.

Let us first look at the Input registers which serve three functions:

1. "Catching" the source operands.
2. Selecting true or complemented values of source operands.
3. "Feeding " operands to the Boolean network.

To explain the presence of one KU module among fourteen GA's, consider that this input register is also used for the Divide and Multiply functional units which use bit 59 to select true or complemented operand values. The two KU modules provide two extra outputs from bit 59 which are used for complement fan-outs, etc.

In deciding to use the true or complemented value of the source operands, the functions performed by the GB modules must be considered. Note that the complemented value of Xj is chosen only if a "Difference" mode bit is received. On the GB module, the gate used to form the exclusive "OR" function is actually an "equivalence" gate. An "exclusive OR" can be performed by "equivalence" if one and only one of the operands is complemented. Study the following examples:

EXCLUSIVE OR:	Using EQUIVALENCE:		
$A = 1100$	$\overline{A} = 0011$		$\underline{A} = 1100$
$B = \underline{1010}$	$B = \underline{1010}$	or	$\overline{B} = \underline{0101}$
$C = 0110$	$C = 0110$		$C = 0110$

In this instance, the complement of Xj is used since the complement of Xk uniquely defines instruction 14 - 17.

The logical function performed in the GB modules is unrelated to the decision to select the true or complemented value of operand Xk. Since for instructions 14 - 17 the "Complement" mode bit is received, the false value of Xk will always be used. (Instructions 14 - 17 specify in their definitions the use of Xk complemented.)

BOOLEAN FUNCTIONAL UNIT REFERENCE CHART

CODE	NAME	FUNCTION	MODE BITS					GB MODULE					
			SUM	PROD	XFER	COMP	DIFF	P3	P5	A	B	C	D
10	TRANSMIT	$Xj \rightarrow Xi$			X			$\overline{Xj}$	$\overline{Xk}$				X
11	LOGICAL PRODUCT	$Xj \cdot Xk$		X				$\overline{Xj}$	$\overline{Xk}$			X	
12	LOGICAL SUM	$Xj + Xk$	X					$\overline{Xj}$	$\overline{Xk}$	X			X
13	LOGICAL DIFFERENCE	$Xj \not\vee Xk$					X	Xj	$\overline{Xk}$		X	X	
14	TRANSMIT	$\overline{Xk} \rightarrow Xi$			X	X		$\overline{Xj}$	Xk	X			
15	LOGICAL PRODUCT	$Xj \cdot \overline{Xk}$		X		X		$\overline{Xj}$	Xk			X	
16	LOGICAL SUM	$Xj + \overline{Xk}$	X			X		$\overline{Xj}$	Xk	X			X
17	LOGICAL DIFFERENCE	$Xj \not\vee \overline{Xk}$				X	X	Xj	Xk		X	X	

Figure 1.7.1-6

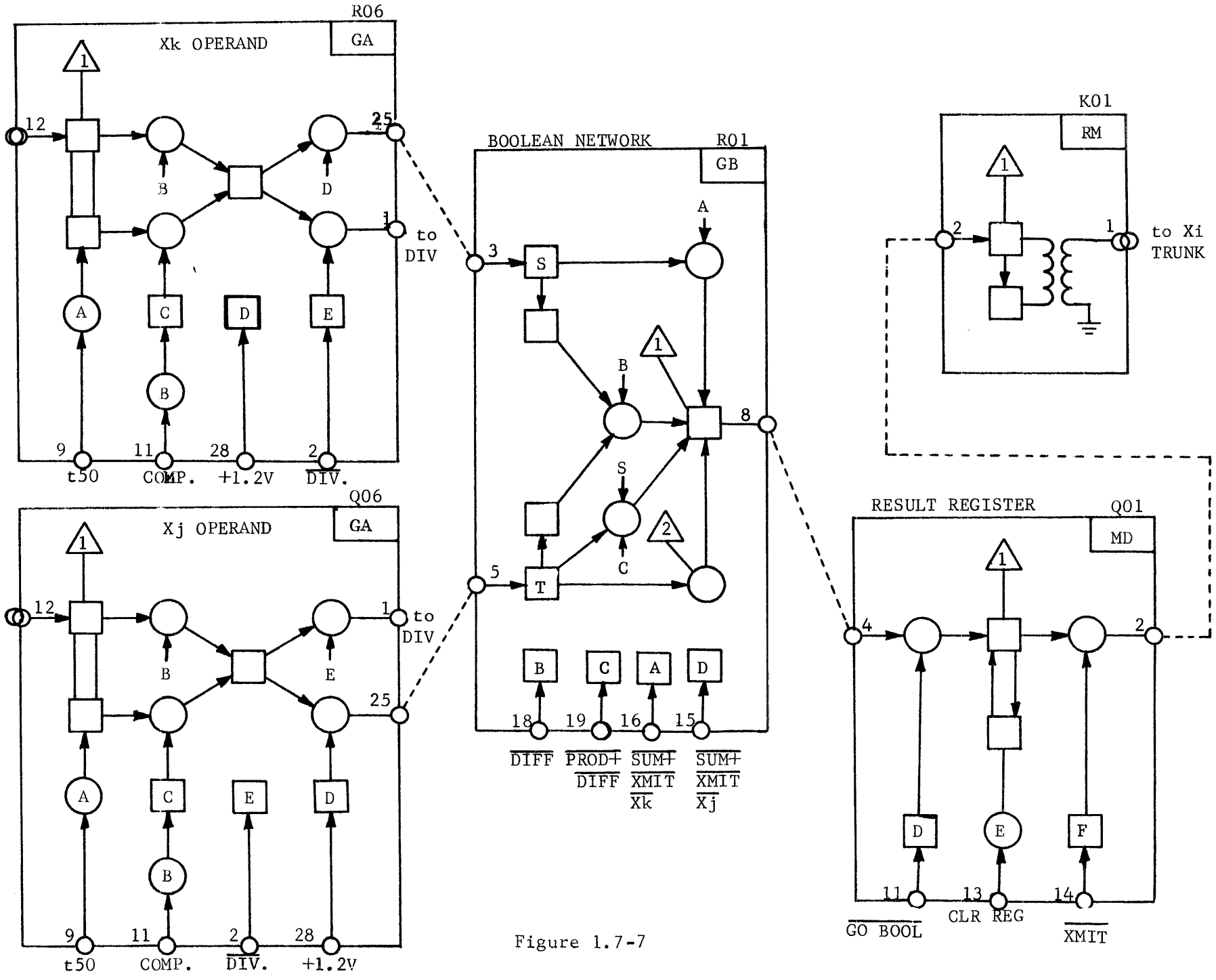

R06
Xk OPERAND
GA
12
25
B
D
1
to
DIV
A
C
D
E
B
9
11
28
2
t50
COMP.
+1.2V
DIV.
Q06
Xj OPERAND
GA
12
1
to
DIV
B
E
25
A
C
E
D
B
9
11
2
28
t50
COMP.
DIV.
+1.2V
BOOLEAN NETWORK
R01
GB
A
3
S
B
1
8
S
5
T
C
2
B
C
A
D
18
19
16
15
DIFF
PROD+ DIFF
SUM+ XMIT Xk
SUM+ XMIT Xj
K01
RM
1
2
1
to Xi
TRUNK
RESULT REGISTER
Q01
MD
1
4
2
D
E
F
11
13
14
CLR REG
GO BOOL
XMIT

Figure 1.7-7

The output of the GA module flip-flop is always enabled to the Boolean Unit via term "D" and pin 25. Note also, that pin 25 will always be the complement of the true or false side of the flip-flop (as selected by the Difference and Complement mode bits) since three inversions take place.

Continuing to the GB modules, notice that the two input pins, 3 and 5, (Xk and Xj, respectively) will always see the complement of the operand (true or false). (Refer to the chart on page 7.1.15.) Thus, the output of the inverters fed by pins 3 and 5 will be the true value of the selected bit state.

Pin 8, the output of the GB module, will represent the true value of the generated result. Thus, any zero into test point 1 will indicate a result of "one" for that bit position. Entering the GB modules are four enables (A, B, C and D) which result from logically combining translations of the mode bits. (See the discussion headed Mode Bits.)

During the Transmit Xj (fm = 10) instruction, only term D will be a logical 1. This condition will enable only the output of inverter T (a reflection of the Xj operand) to the result register via Test Points 2 and 1. (Terms A, B and C are zeros, disabling the other three inputs to TP1.) Thus, Xj is sent to the result register.

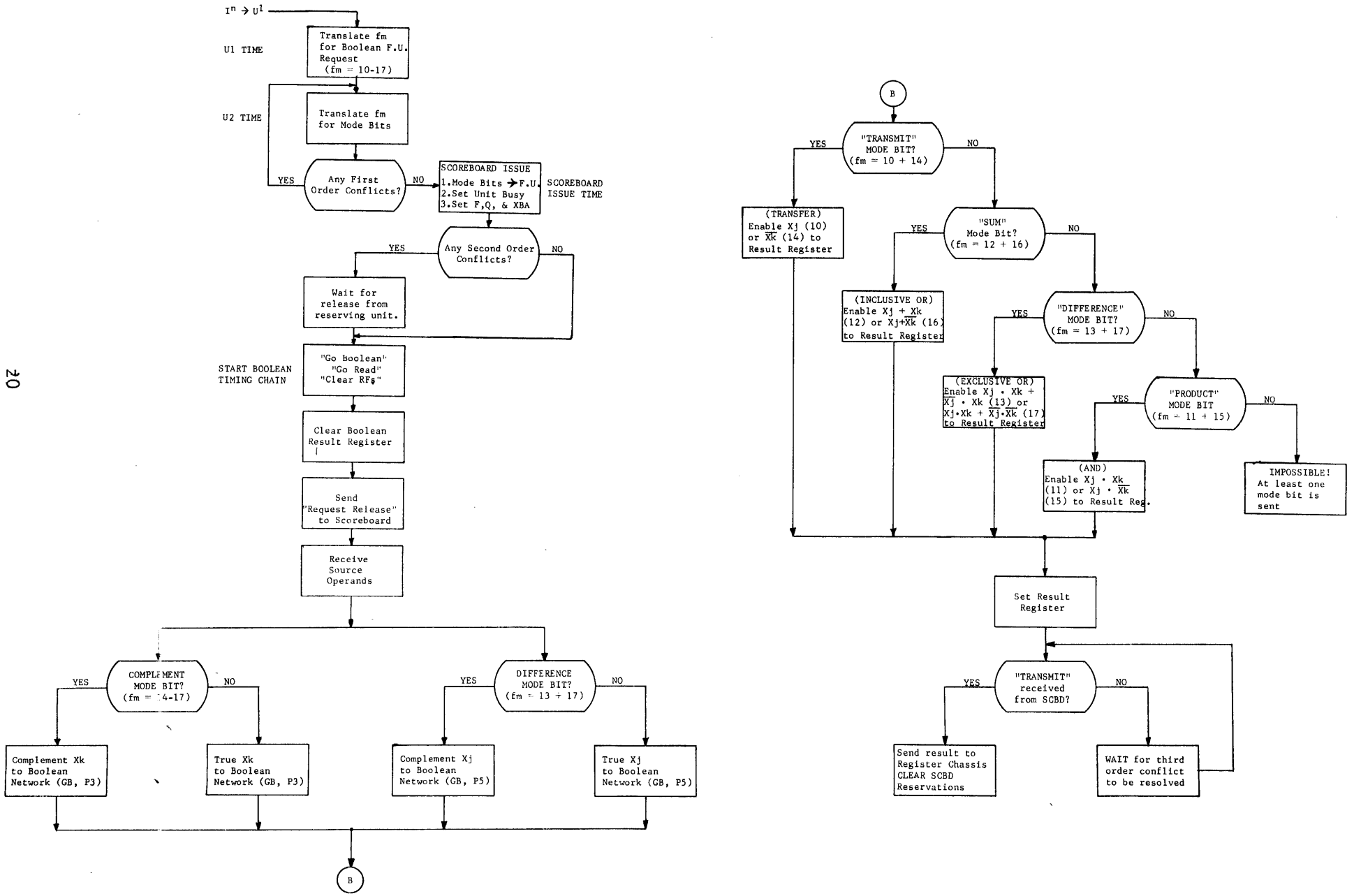
During the Transmit $\overline{Xk}$ instruction (fm = 14), only term A will be a logical 1. This term enables the output of inverter S to pin 8. Inverter S is a reflection of Xk complemented since the "Complement" mode bit is received with a 14 instruction. Again, all other inputs to TP1 are disabled because terms B, C and D are logical zeros. Thus. Xk is sent to the result network.

Instructions 12 (Xj + Xk to Xi) and 16 (Xj + $\overline{Xk}$ to Xi) are, in essence, the combination of the Transmit instructions with the "Complement" mode bit distinguishing one from the other. Both A and D are enabled. Consequently, inverter S or T or both are gated to the output network (Inclusive OR). T again reflects the Xj operand. During the 12 instruction, S reflects Xk and pin 8 sees Xj + Xk. During the 16 instruction, S reflects Xk complemented and pin 8 sees Xj + $\overline{Xk}$.

During the two logical product instructions (11 and 15) terms A, B and D are disabled and only term C is a logical 1. Term C enables the gate which ANDs inverters S and T. Thus, during instruction 11, Xj · Xk are sent to the output network. For instruction 15, the "Complement" mode bit results in Xj · $\overline{Xk}$ being sent to the output network.

During the Logical Difference instructions (13 and 17), remember the complement of the Xj operand is selected. Thus, the output of inverter T reflects $\overline{Xj}$. During the 13 instruction, S reflects Xk. Since term C once again enables the ANDing of inverters S and T, we obtain $\overline{Xj}$ · Xk (half of the desired Exclusive OR) at pin 8. Term B, also enabled during Difference, gates the complement of T and S to the output network. We thus obtain Xj · $\overline{Xk}$ at pin 8 - the other half of the Exclusive OR function. Thus, the result of the 13 instruction is $\overline{Xj}$ · Xk or Xj · $\overline{Xk}$. The 17 instruction differs from the 13 only because the "Complement" mode bit is once again present. Thus, inverter S reflects $\overline{Xk}$ and the AND gates enabled by B and C form Xj · Xk and $\overline{Xj}$ · $\overline{Xk}$ respectively. Either of these conditions make pin 8 a logical 1.

BOOLEAN FUNCTIONAL UNIT - FLOW CHART



7.1.6 GLOSSARY OF LOGICAL TERMS

1. AND: A logical relation between two propositions which holds true only if both propositions are true.

   Example: 1100 SYMBOL = ·
   1010
   1000

2. COMPLEMENT: A number which, when added to a quantity, will result in a sum which is the modulus of the number system.

   Example: Complement 0100 SYMBOL: Comp X = $\overline{X}$
   Quantity 1011
   Sum 1111

3. EQUIVALENCE: A logical relation between two propositions which holds true only if both propositions are true or both are false. (Also the complement of EXCLUSIVE "OR")

   Example: 1100 SYMBOL: ≡
   1010
   1001

4. EXCLUSIVE "OR": A logical relation between two propositions which holds true only if the first proposition is true and the second false or the second is true and the first is false.

   Example: 1100 SYMBOL: ⊻
   1010
   0110

5. IMPLICATION: A logical relation between two propositions which is false only if the first proposition is true and the second is false.

   Example: 2nd 1100 SYMBOL: ⊃
   1st 1010
   1101

6. INCLUSIVE "OR": A logical relation between two propositions which holds true only when either or both propositions are true.

   Example: 1100 + 1010 = 1110 SYMBOL: +

7. Logical DIFFERENCE: Same as EXCLUSIVE "OR".

8. Logical PRODUCT: Same as "AND".

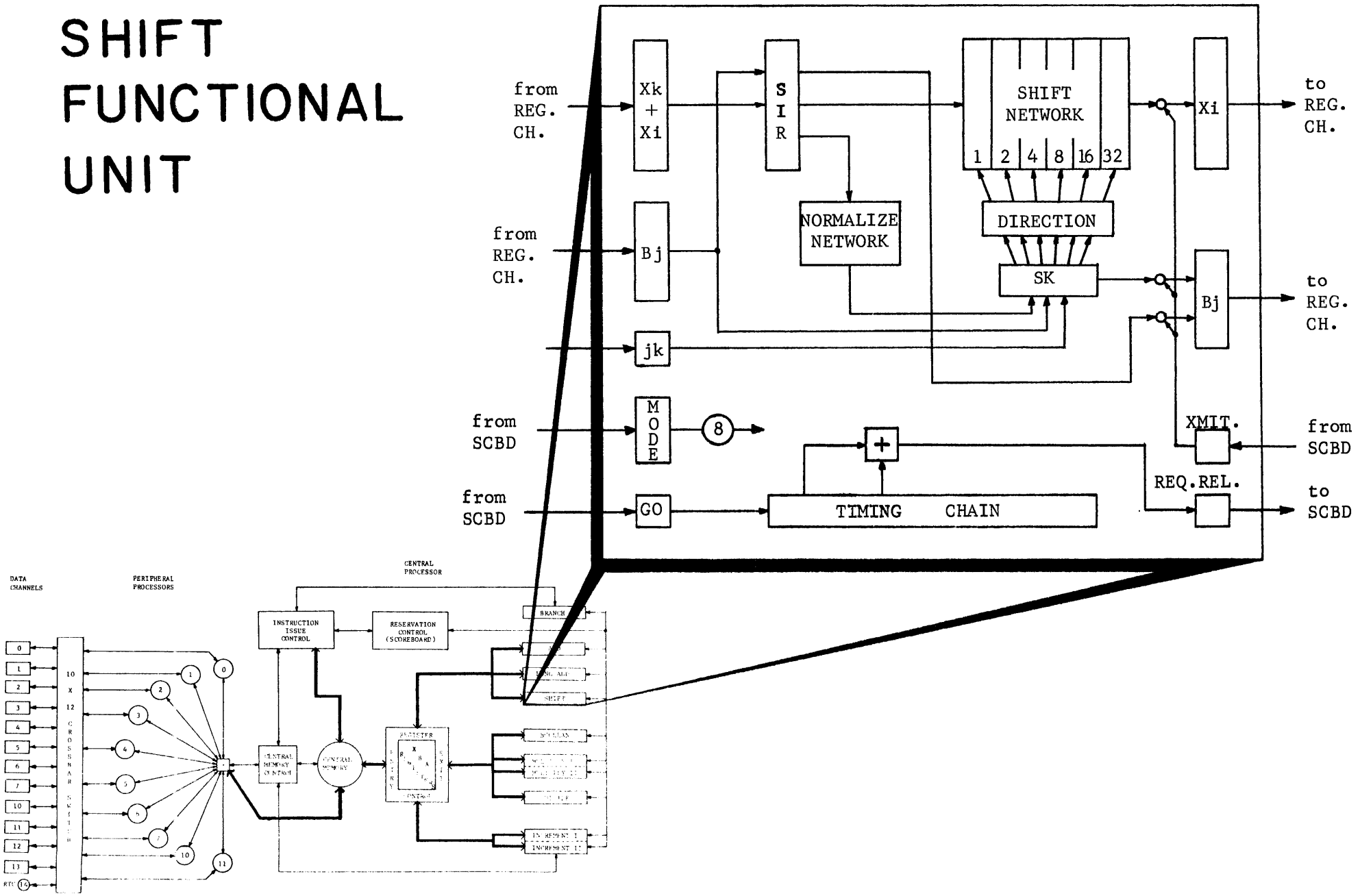
9. Logical SUM: Same as INCLUSIVE "OR".

SECTION 7.2

SHIFT

FUNCTIONAL UNIT

SHIFT FUNCTIONAL UNIT



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:

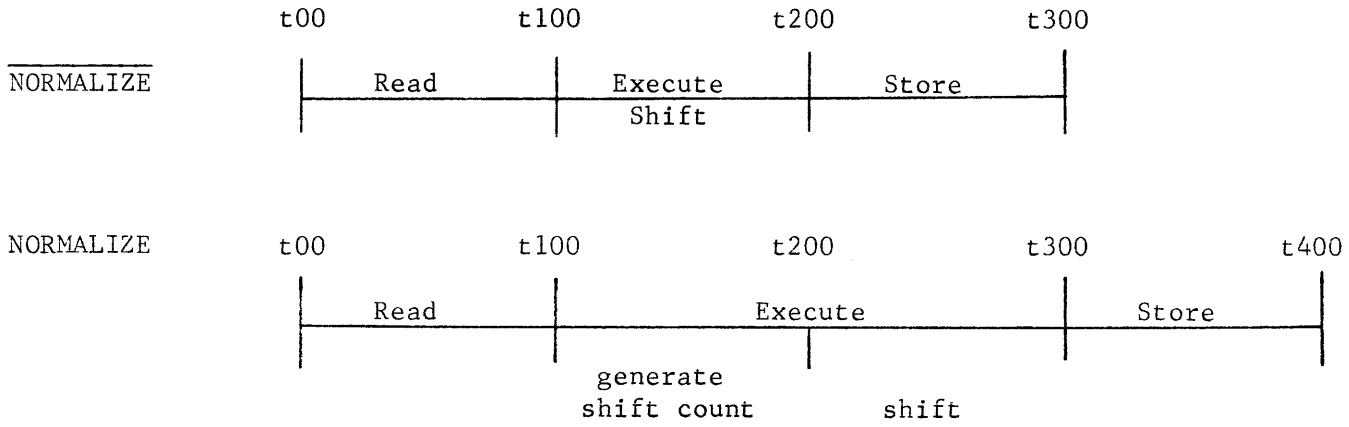


FIGURE 7.2.1

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 normalize shift count from the original operand (Xk) 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 FLOW

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)

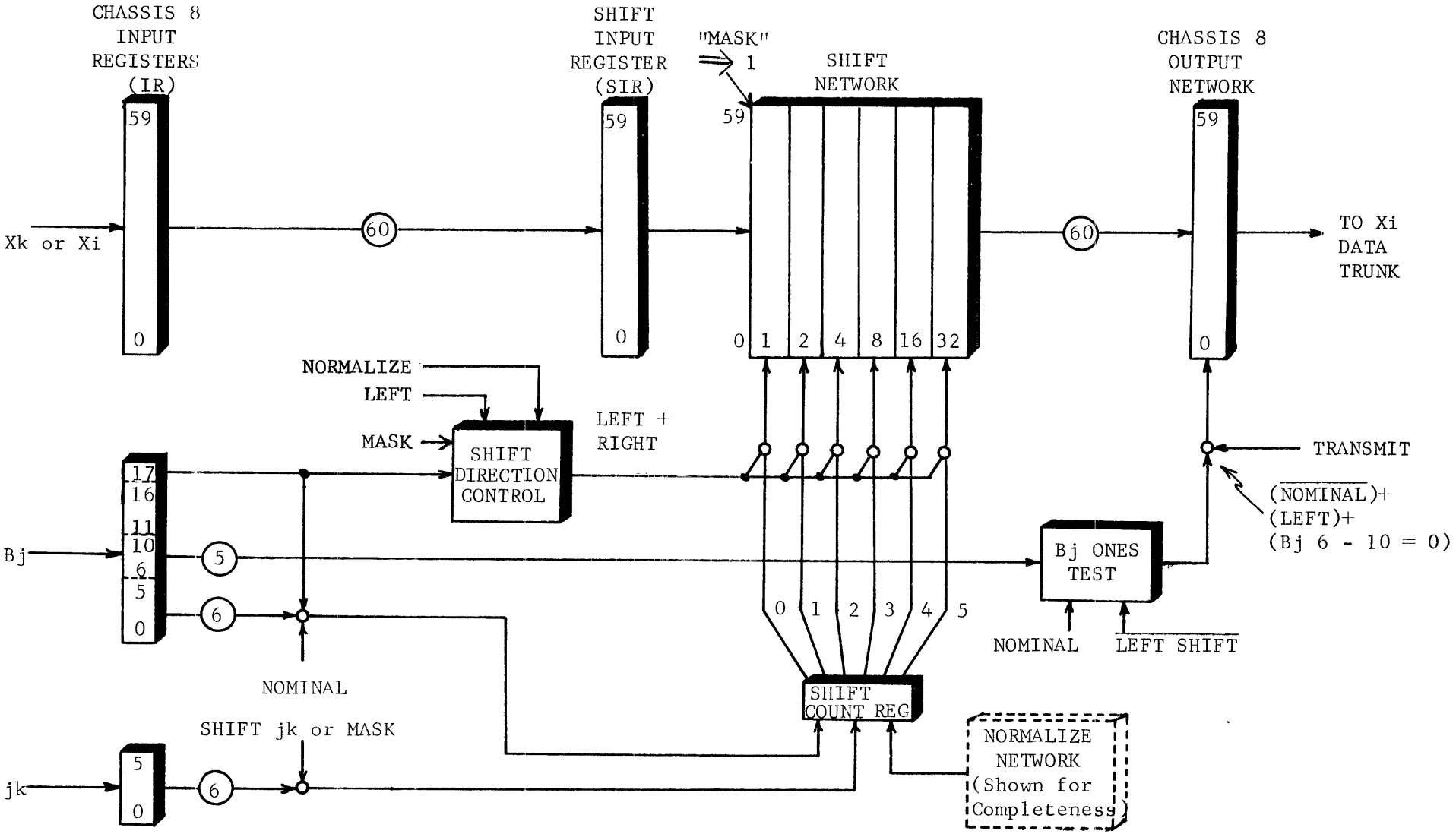


FIGURE 7.2-2

22 SHIFT Xk NOMINALLY LEFT Bj places to Xi (LXi = Bj, Xk)

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 Bj sign (bit 17) is positive, the shift is left circular; if Bj 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 Bj 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 Xk NOMINALLY RIGHT Bj places to Xi (AXi = Bj, Xk)

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 Bj sign (bit 17) is positive, the shift is right (end-off with sign extension);* if Bj 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 Bj 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 Xi.

24 NORMALIZE Xk in Xi and Bj (NXi, Bj = Xk)

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 Xk 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 Xk 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 Xk exponent equals 1777 or 3777. The output of bits 0 through 47 of the shift network are gated (complemented if Xk 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.

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

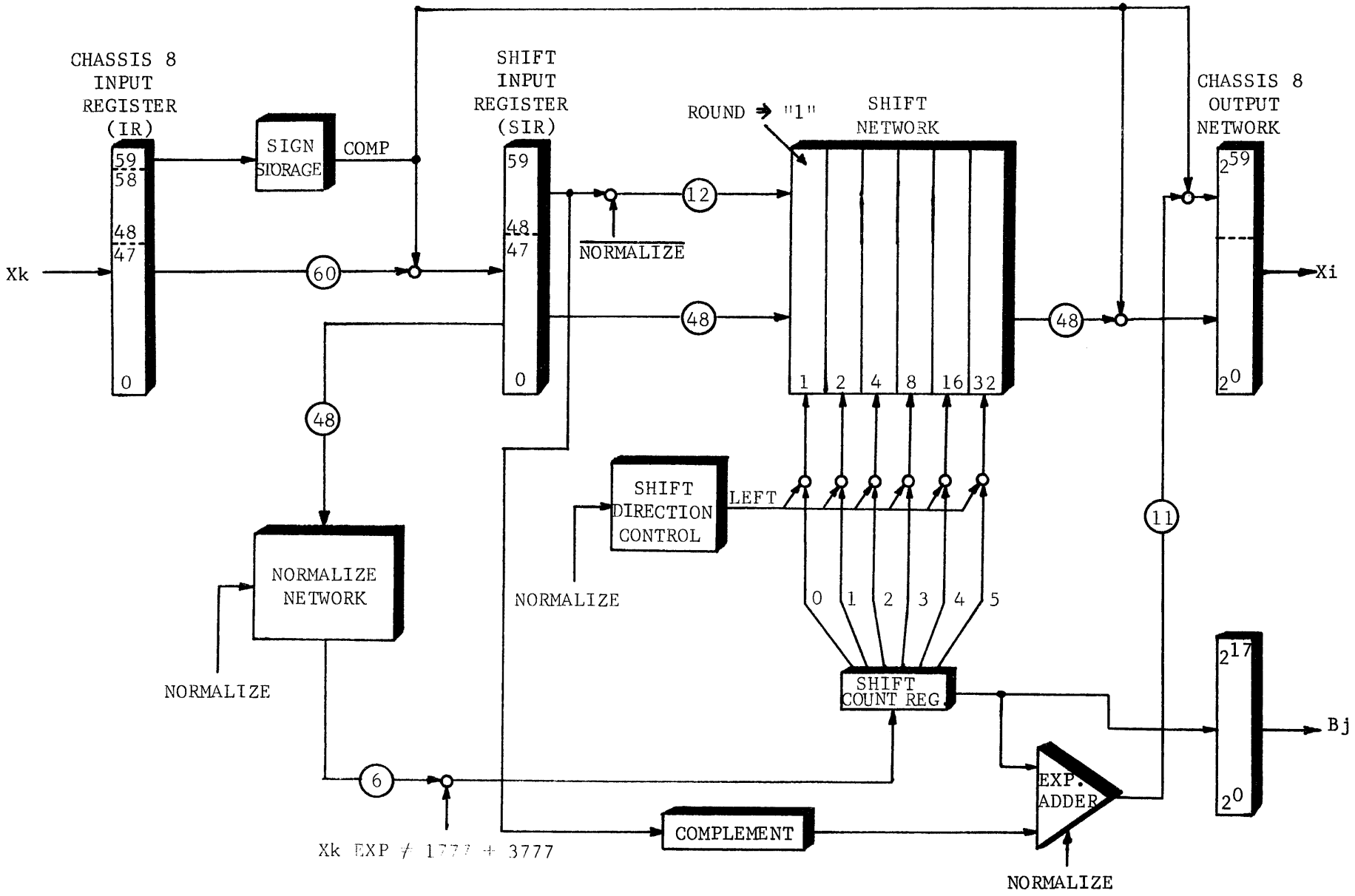


FIGURE 7.2-3

25 ROUND and NORMALIZE Xk in Xi and Bj (ZXi, Bj = Xk)

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 Xk is in infinite or indefinite form, it is sent out intact 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 into the least significant bit, thus adding $\frac{1}{2}$. It is important to note that rounding does not occur if the coefficient is already normalized.

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

26 UNPACK Xk to Xi and Bj (UXi, Bj = Xk)

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 registed j.

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

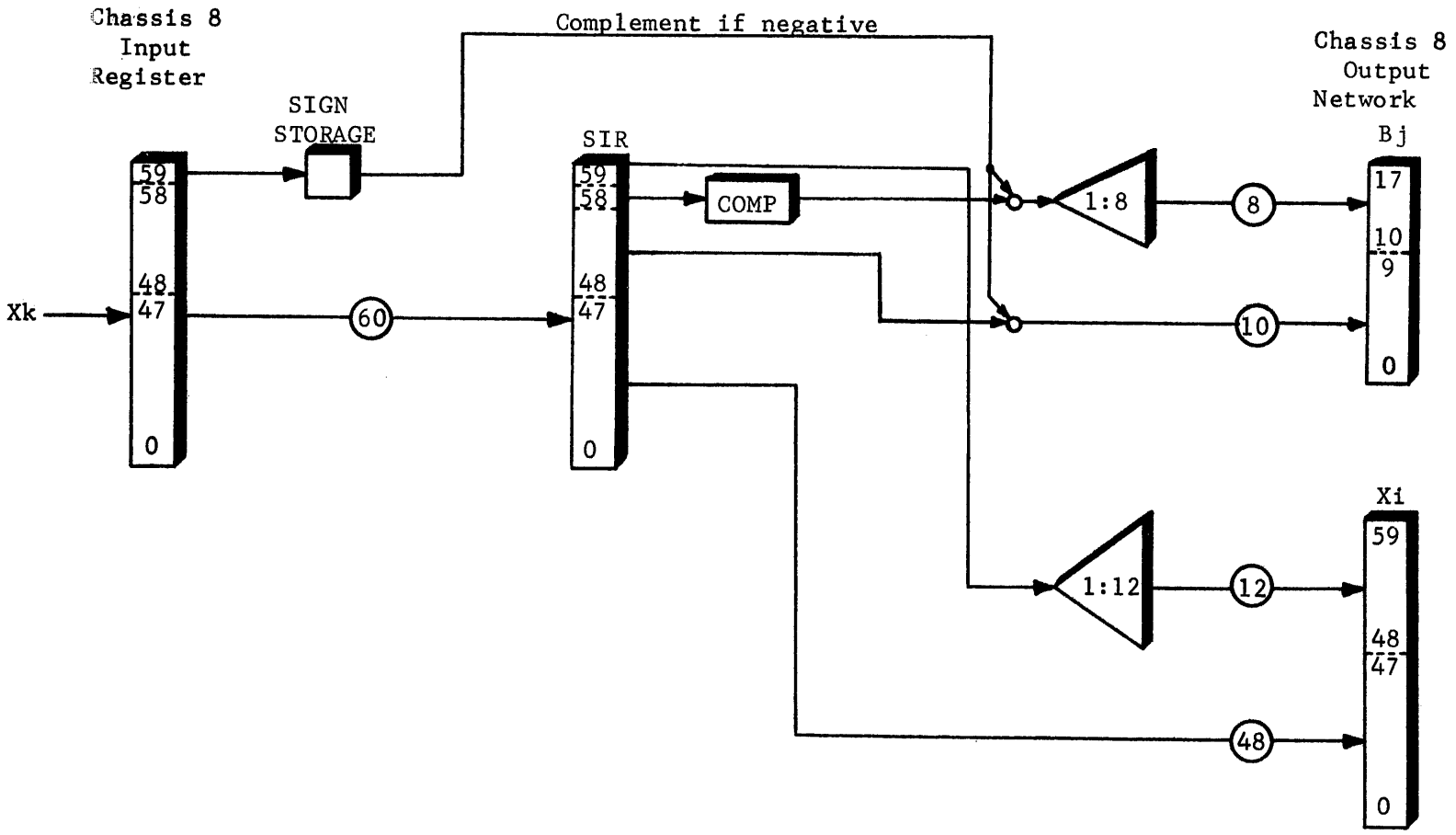


FIGURE 7.2-4

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 Xi output network. The coefficient sign is extended to bits 48-59 through a fan-out. Bits 48-57 of SIR 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 fannedout to bits 10-17 of the Bj output network. These 18 bits will be completed into 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 (PXi = Bj, Xk)

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 Z 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 block diagram #1, Figure 7.2-2.

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

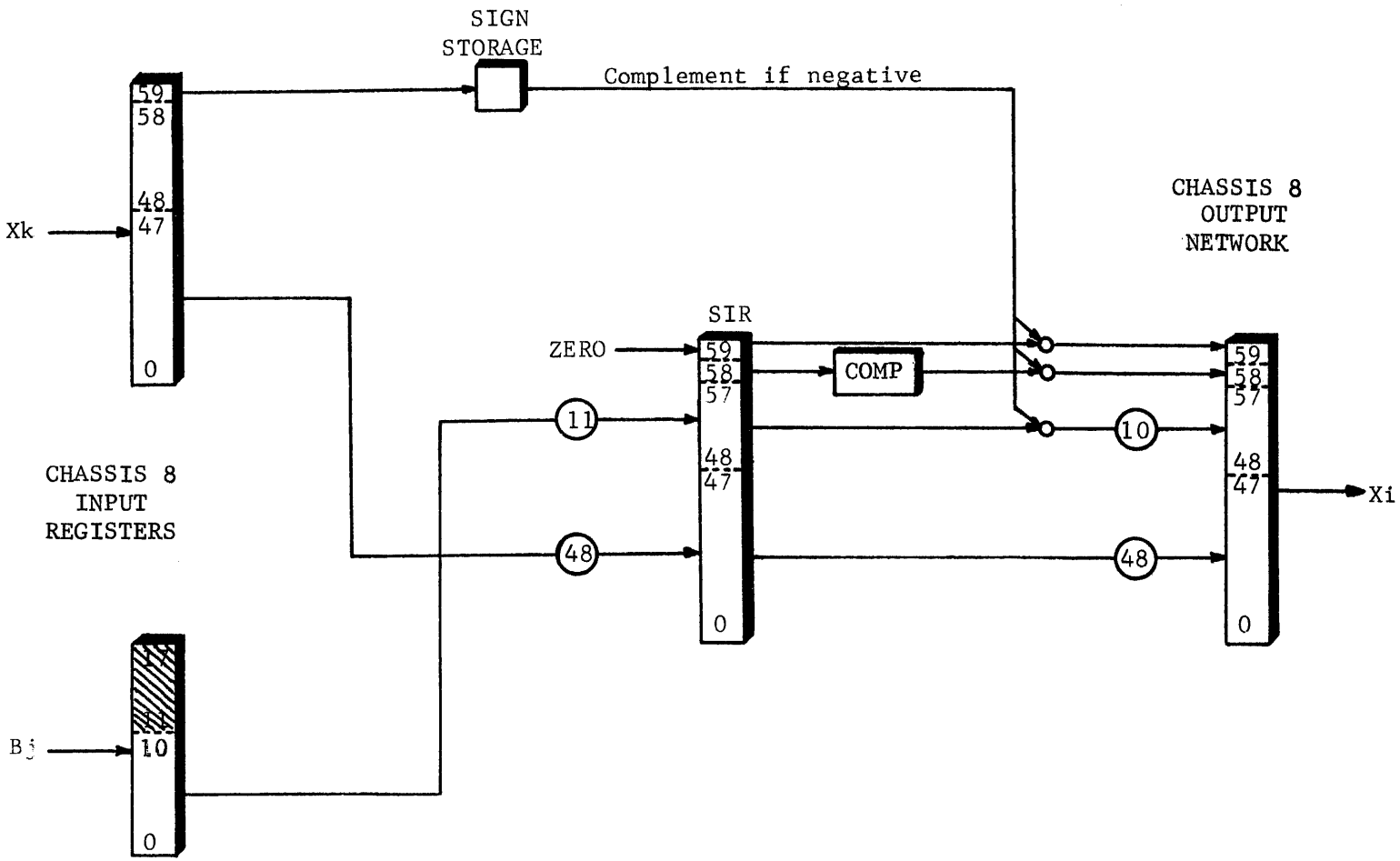


FIGURE 7.2-5

43 FORM jk MASK in Xi (MXi jk)

DEFINITION: This instruction forms a mask in X register i. The 6-bit quantity jk defines the number of ones in tne 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 Xi 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	Xi	Xi	300
21	Shift Right	jk	Xi	Xi	300
22	Shift Left Nominally	Bj(bits 17, 5-0)	Xk	Xi	300
23	Shift Right Nominally	Bj(bits 17, 5-0)	Xk	Xi	300
24	Normalize	Normalize Network	Xk	Xi and Bj	400
25	Round and Normalize	Normalize Network	Xk	Xi and Bj	400
26	Unpack	None	Xk	Xi and Bj	300
27	Pack	None	Xk and Bj	Xi	300
43	Mask	jk	None	Xi	300

FIGURE 7.2-6

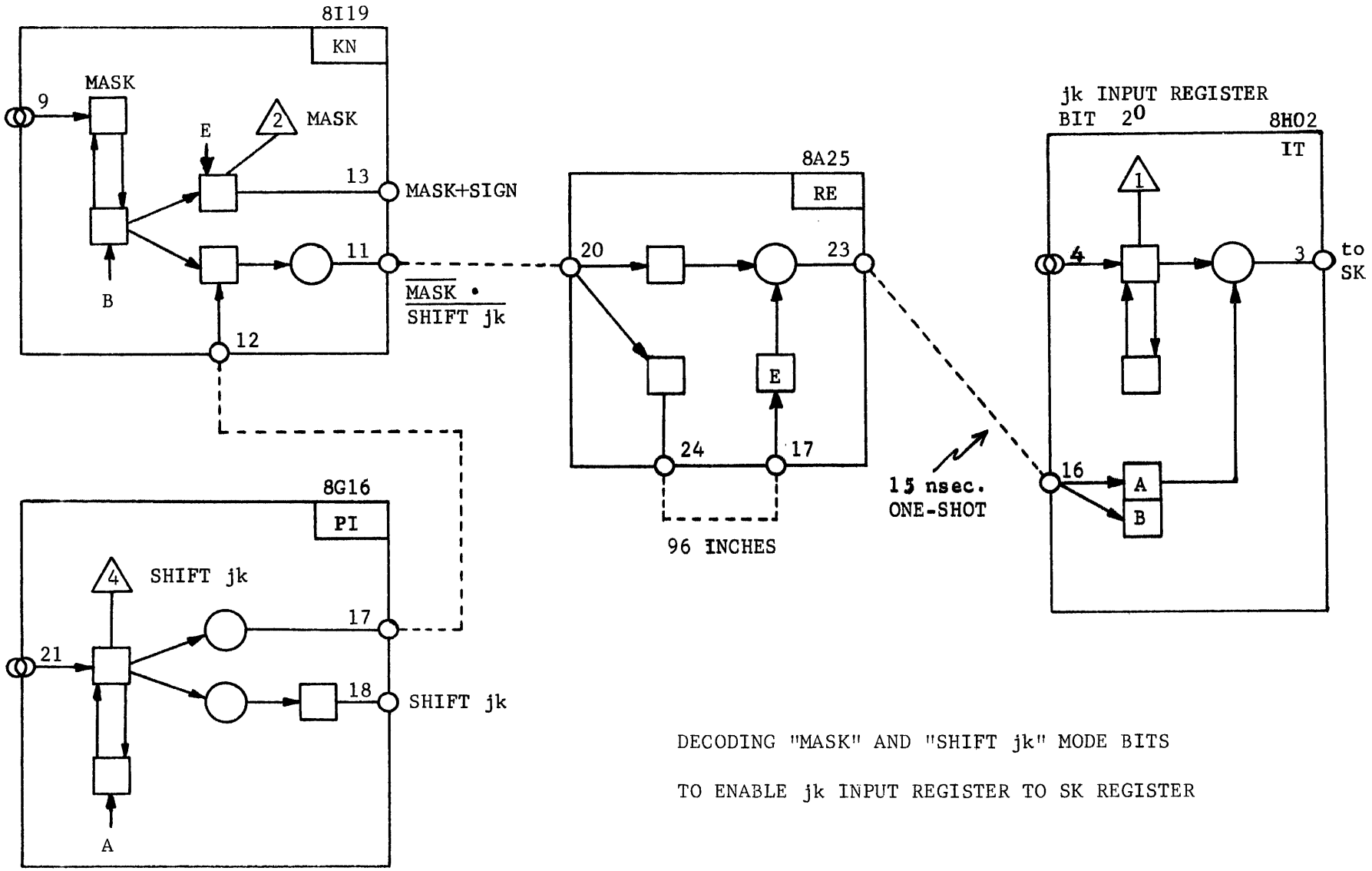
Note that two instructions (20 & 21) specify Shift jk, two (20 & 22) specify Shift Left, two (21 & 23) specify Shift Right, two (22 & 23) are Nominal Shifts, two (24 & 25) are Normalize instructions, and there is one each of Mask (43), Round (25), Pack (27), and Unpack (26). With this information, the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

MODE BITS	fm (INSTRUCTION)
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

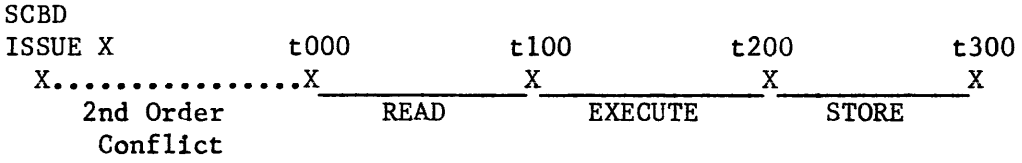
TO ENABLE jk INPUT REGISTER TO SK REGISTER

FIGURE 7.2-8

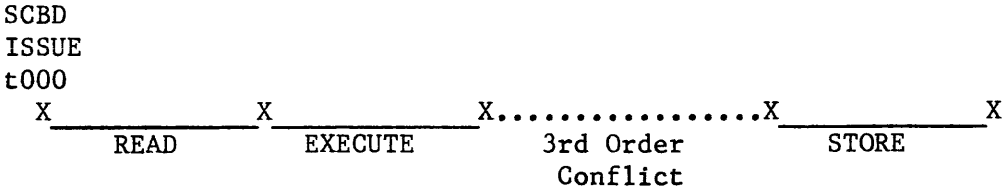
7.2.4 TIMING SEQUENCE

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.

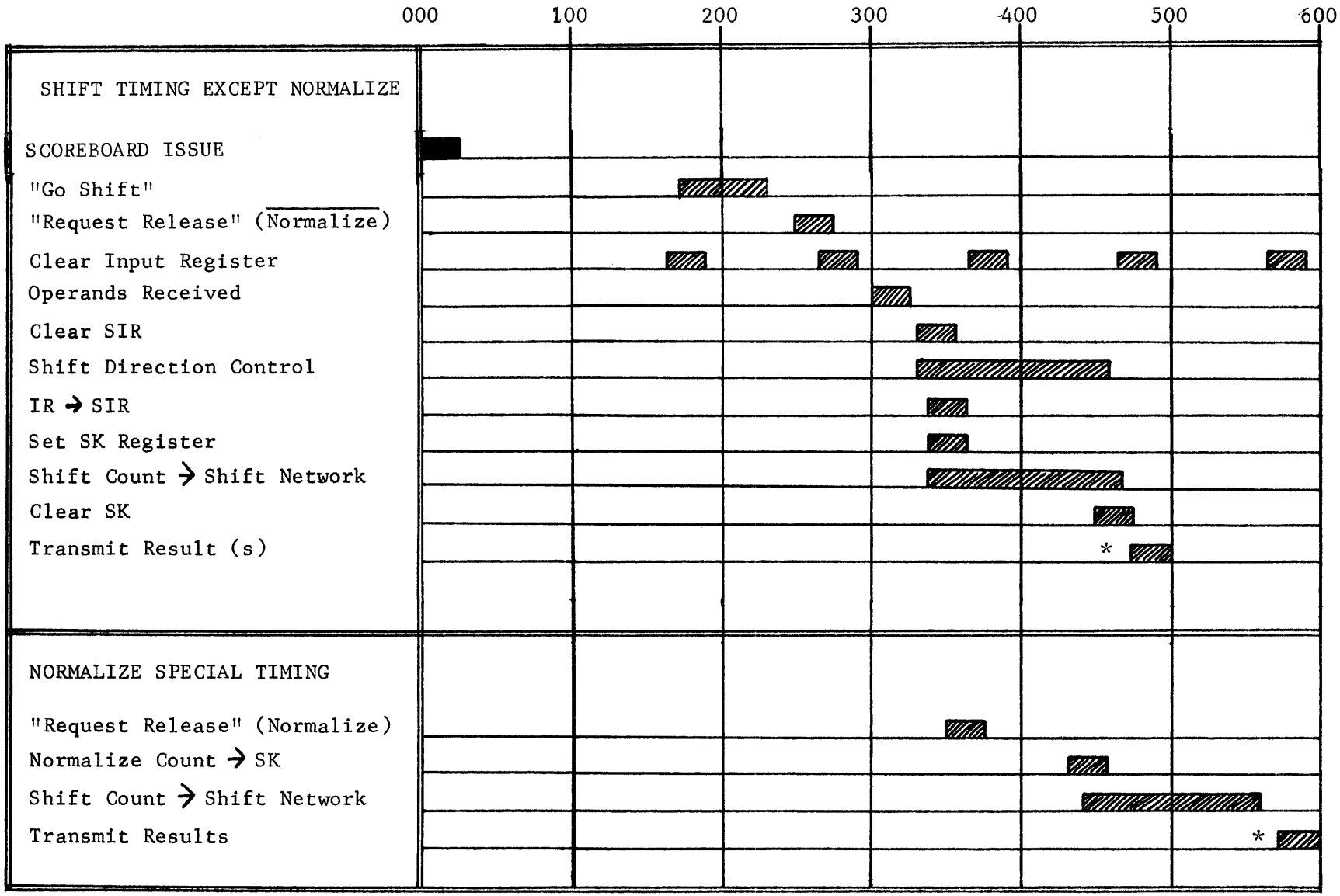


If a third order conflict should occur, gating of the result to the register chassis (Transmit Result) will be delayed for the length of the conflict.



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 $\Longrightarrow$ Left)
2. Shift Right "X" places (if Direction Control $\Longrightarrow$ 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

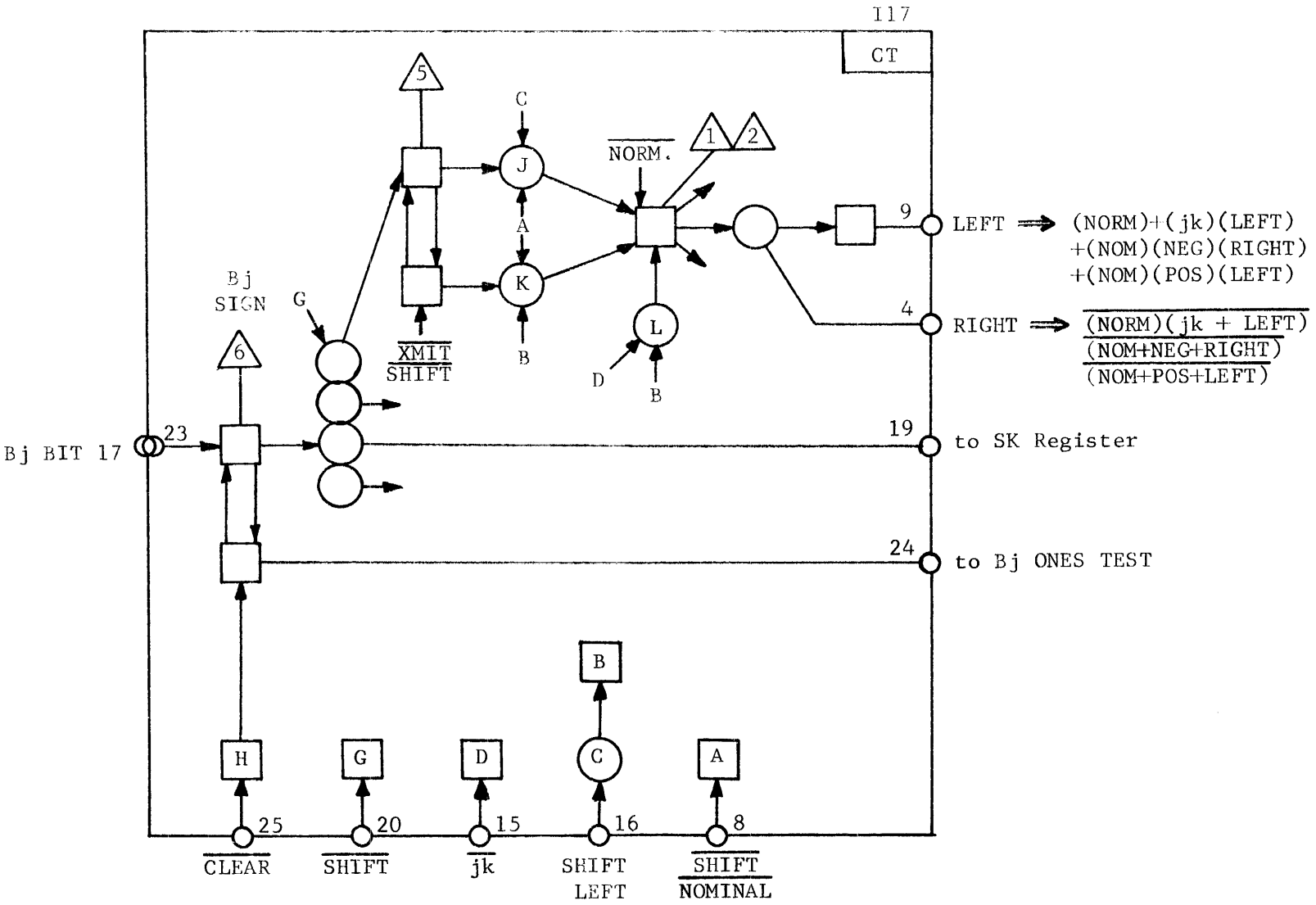


FIGURE 7.2-10

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 (117, 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 shift 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 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 $\overline{\text{NORMALIZE}}$ 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	(NORM) + (jk) (LEFT) + (NOM) (NEG) (RIGHT) + (NOM) (POS) (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

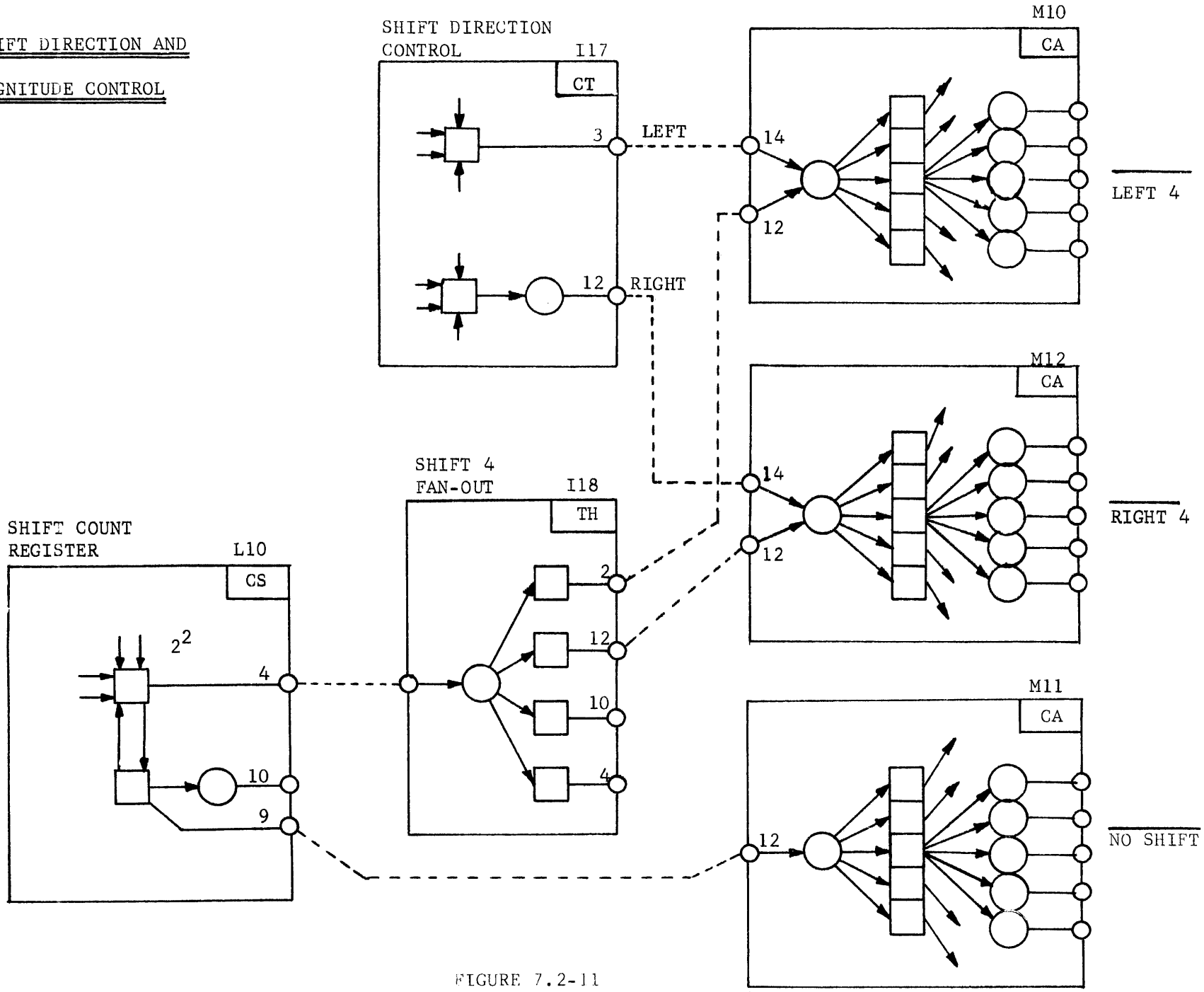


FIGURE 7.2-11

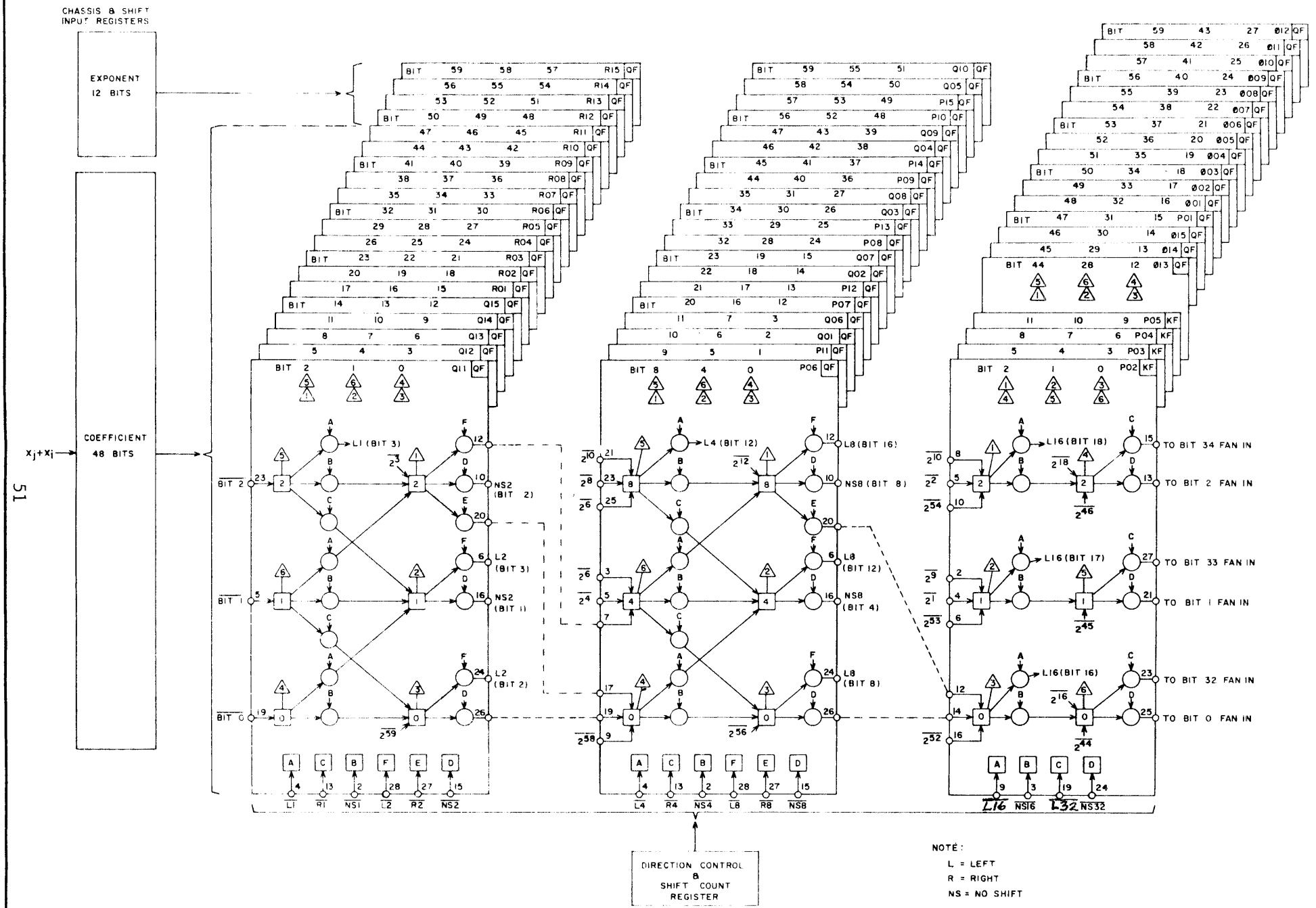
power of two and each "one" bit of the 6-bit shift count in SK gates a corresponding level of the network. A "zero" in any bit position of SK implies no shift (See Figure 7.2-11). Slave inverters on each bit of SK are gated by shift direction (left or right) according to instruction requirements. (Refer 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. (See Figure 7.2-12H).

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.

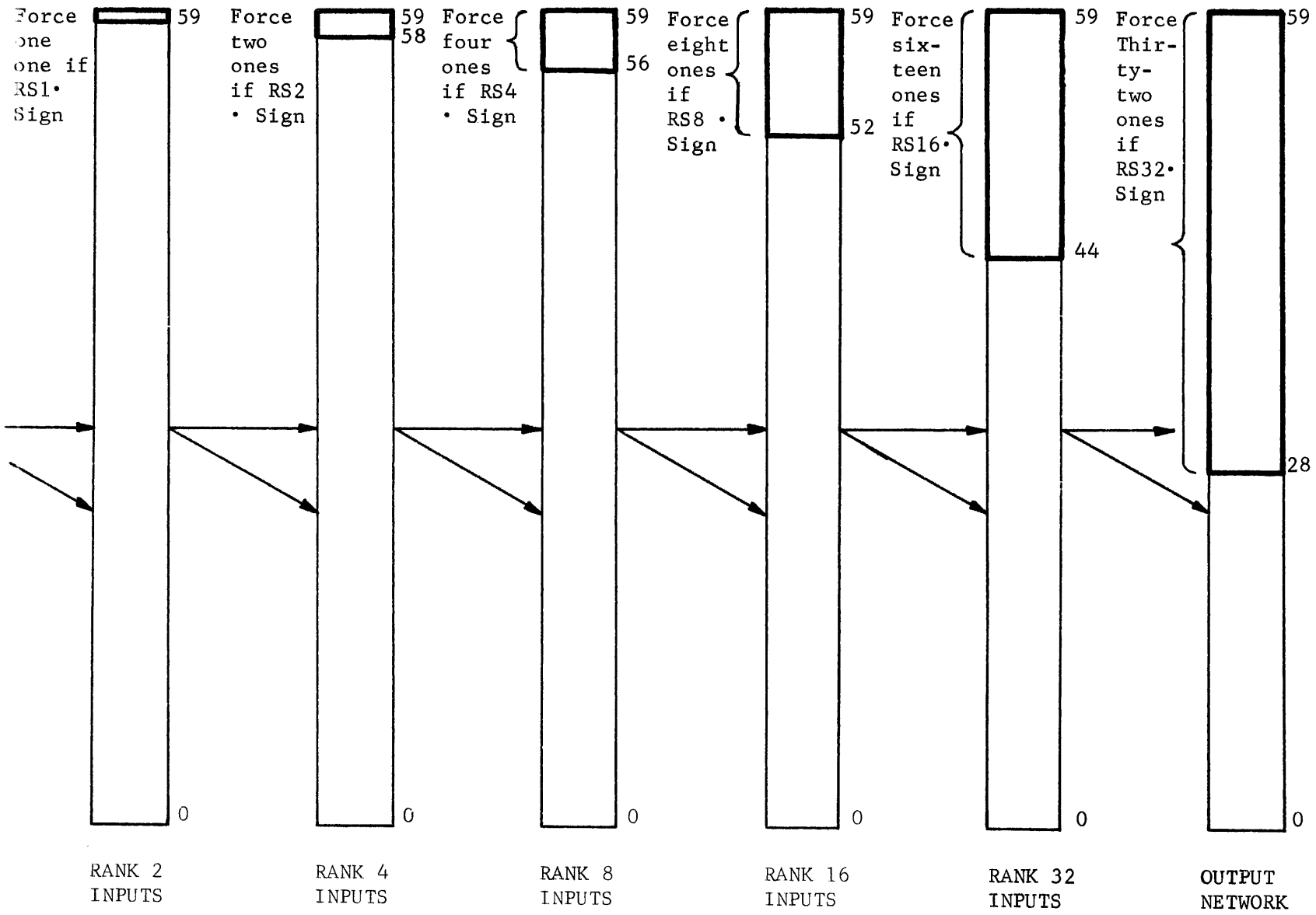


Shift Network

Figure 7.2-12

SIGN EXTENSION BLOCK DIAGRAM

FIGURE 7.2-12A



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 Xk 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

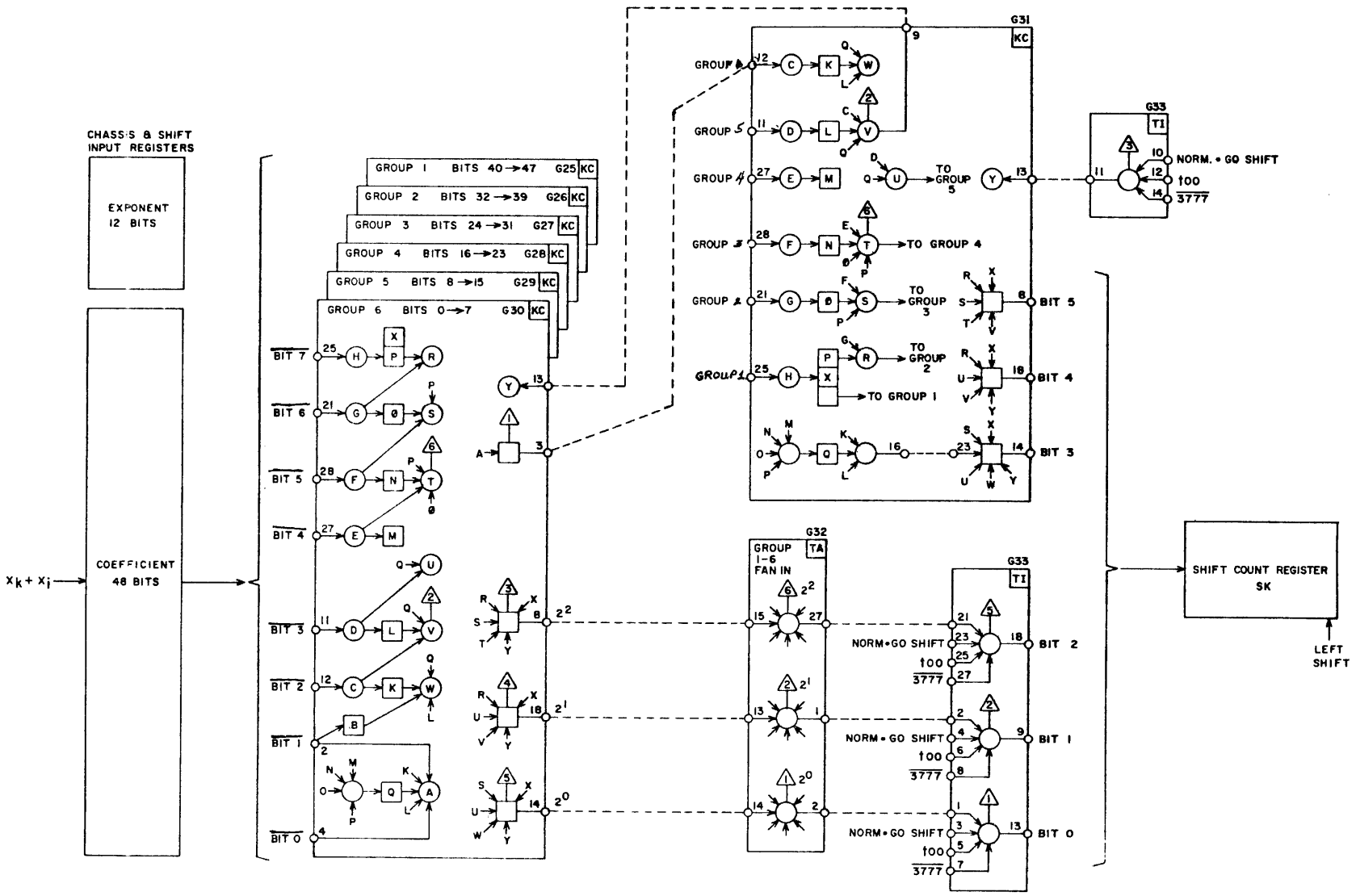


Figure 7.2-13

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 Bj 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 Bj 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 (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⟶ 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 an exponent greater than 10 bits in magnitude to Bj. Consequently, the Shift Nominal instruction, if used properly, should never find and exponent (Bj Registe) greater in magnitude than 10 bits.

Figure 7.2.14 is a logic drawing of the Bj Ones Test Network. The test is 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 sending 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 zero in each bits 6 through 10 (terms "X" and "F",

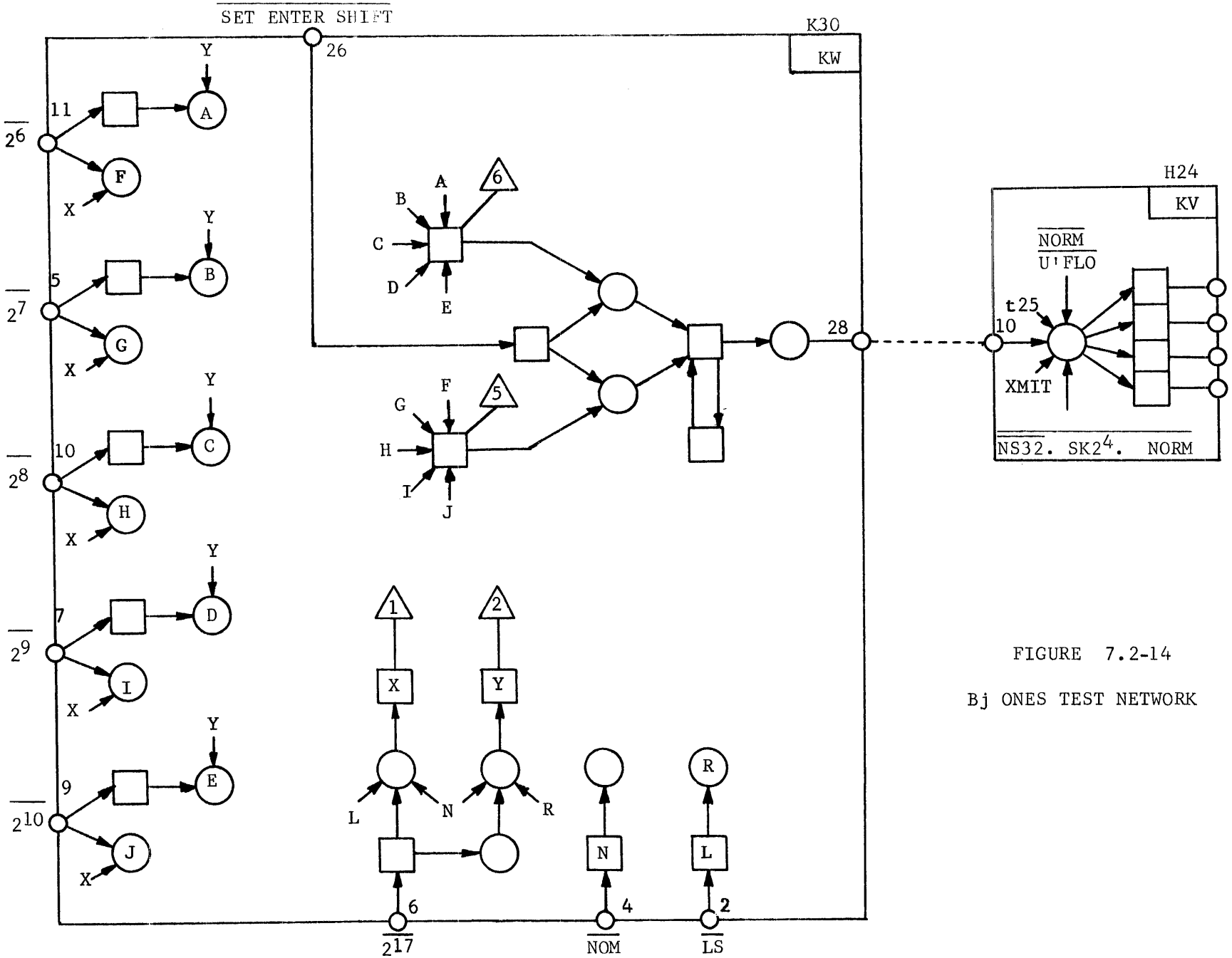


FIGURE 7.2-14

Bj ONES TEST NETWORK

"G", "H", "I", and "J"). If Bj is negative, zeros are checked because the magnitude is in complement form. If any of these bits is a zero when Bj 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 Bj 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 Bj 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 = (NOMINAL) (LEFT) (2^{17}) or, Nominal Left and Bj = Neg (Right Shift)

Y = (NOMINAL ($\overline{\text{LEFT}}$) ($\overline{2^{17}}$) or, Nominal Right and Bj = 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 Xi.

7.2.9 EXPONENT ADDER

Since the coefficient of Xk is shifted left (increased) during the normalize process, the exponent of Xk must be decremented by an equivalent value. This is the function of the exponent adder, i.e. to substract 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:

(Xk) = 2135 0------0327715 (327715×2^{135})

After Normalizing:

(Xi) = 2076 6576320------0 (657632×2^{76})

(Bj) = 0-----037 (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 contain 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 indicates 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	= 51	- 51	15	- 15
RESULT	= 1761	-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)	60	- 60
	0000	-1777

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

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

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

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

Positive, but transition to negative is not possible.

EXP	2060	+60
(SK) Maximum	60	-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

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

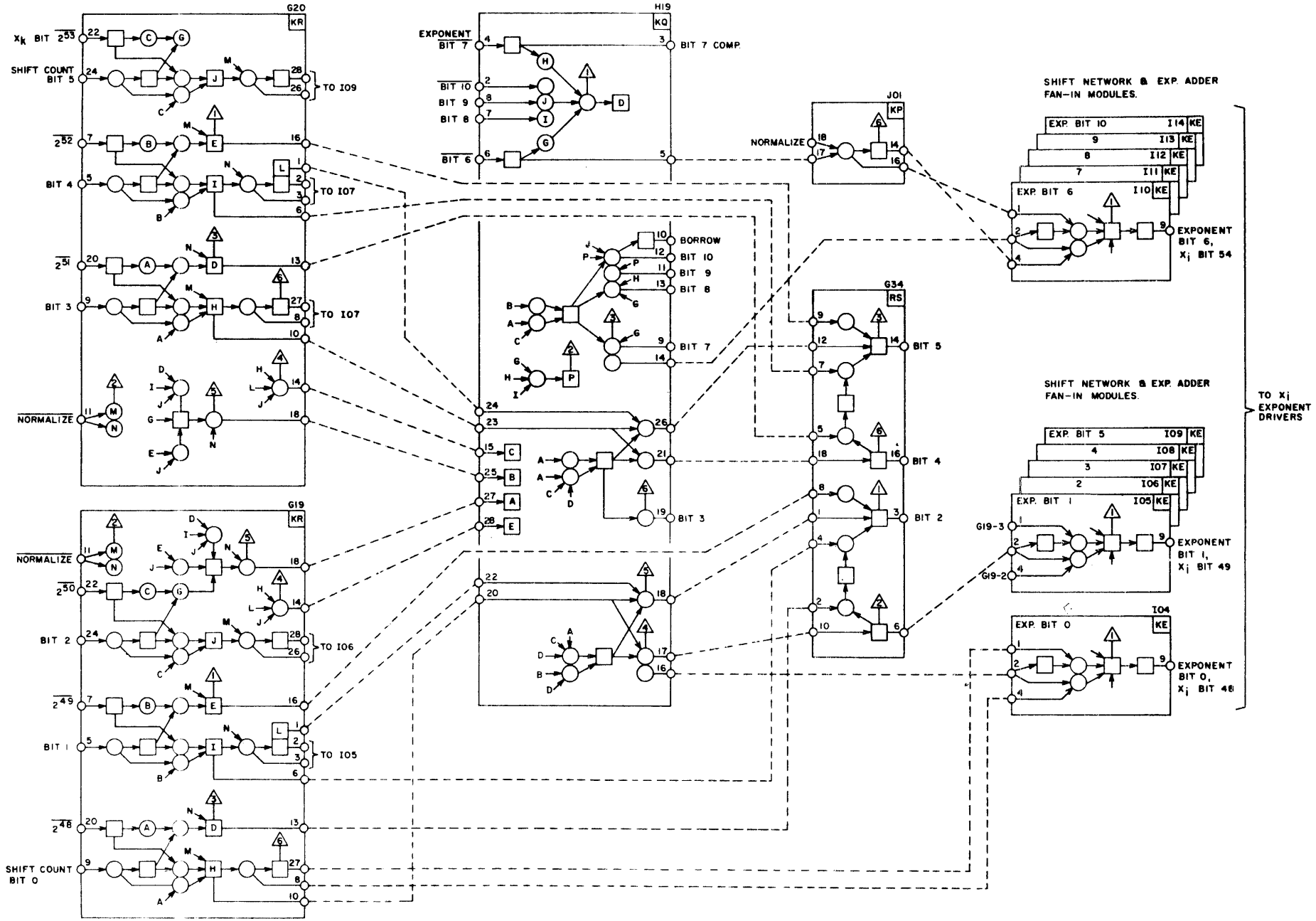
Xk =	2031	=	10	000	011	001	=	+31
Sk =	55 EAB	=			101	101 EAB	=	-55
	1754		01	111	101	100		
	1					1		
	1753	=	01	111	101	011	=	-24

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) $\longrightarrow$ Positive to Negative transition.
2. (EXP bit $2^{10} \neq 1$) (EAB) $\longrightarrow$ 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 or a satisfy is present in bits 9-6 ($\overline{D} \Rightarrow \overline{10} + 9 + 8 + 7 + 6 = \overline{20XX}$) 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)



EXPONENT ADDER

Figure 7.2-17

SHIFT FUNCTIONAL UNIT

FLOW CHART

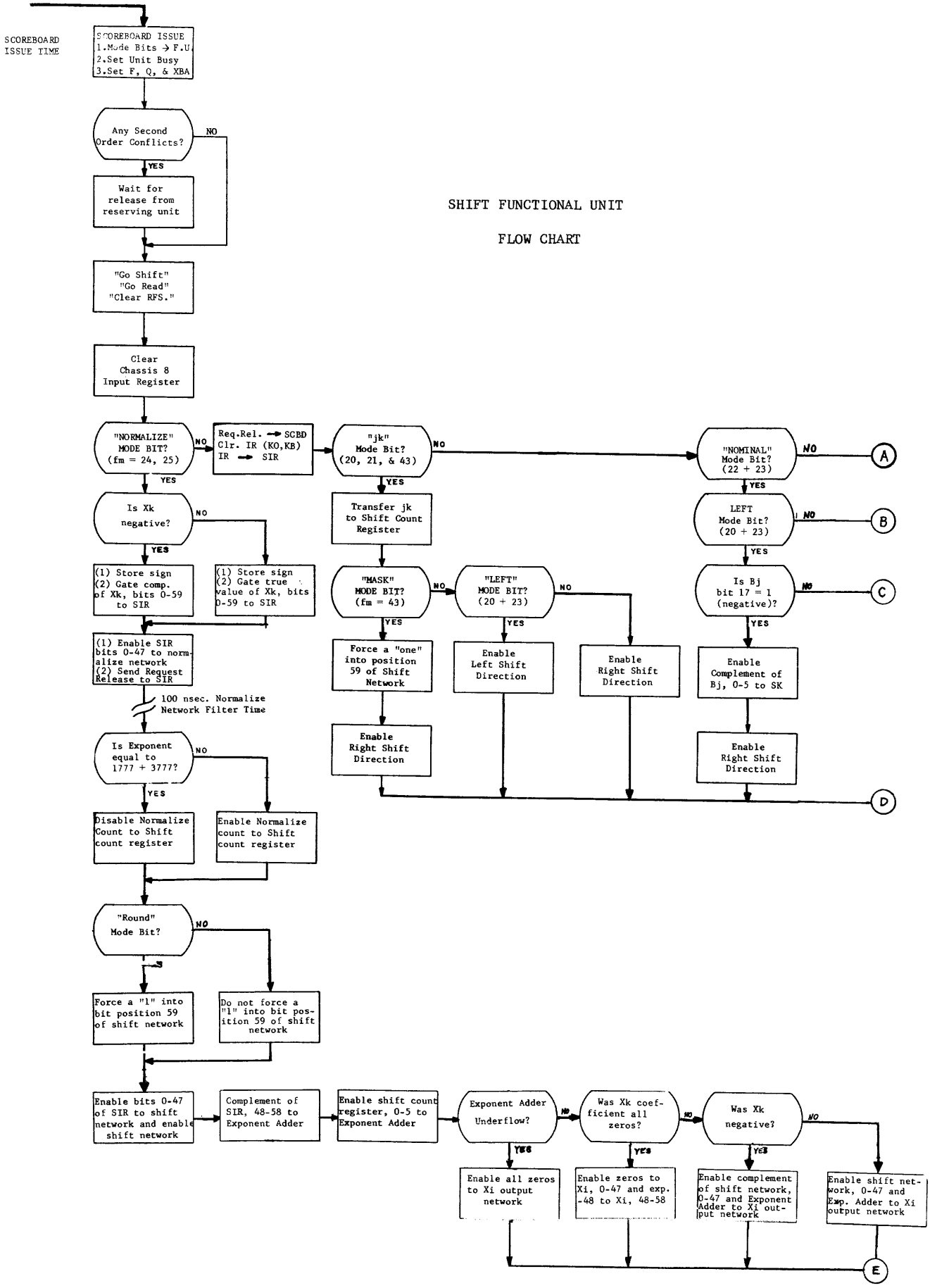


Figure 7.2-18A

SHIFT FUNCTIONAL UNIT

FLOW CHART

(CONTINUED)

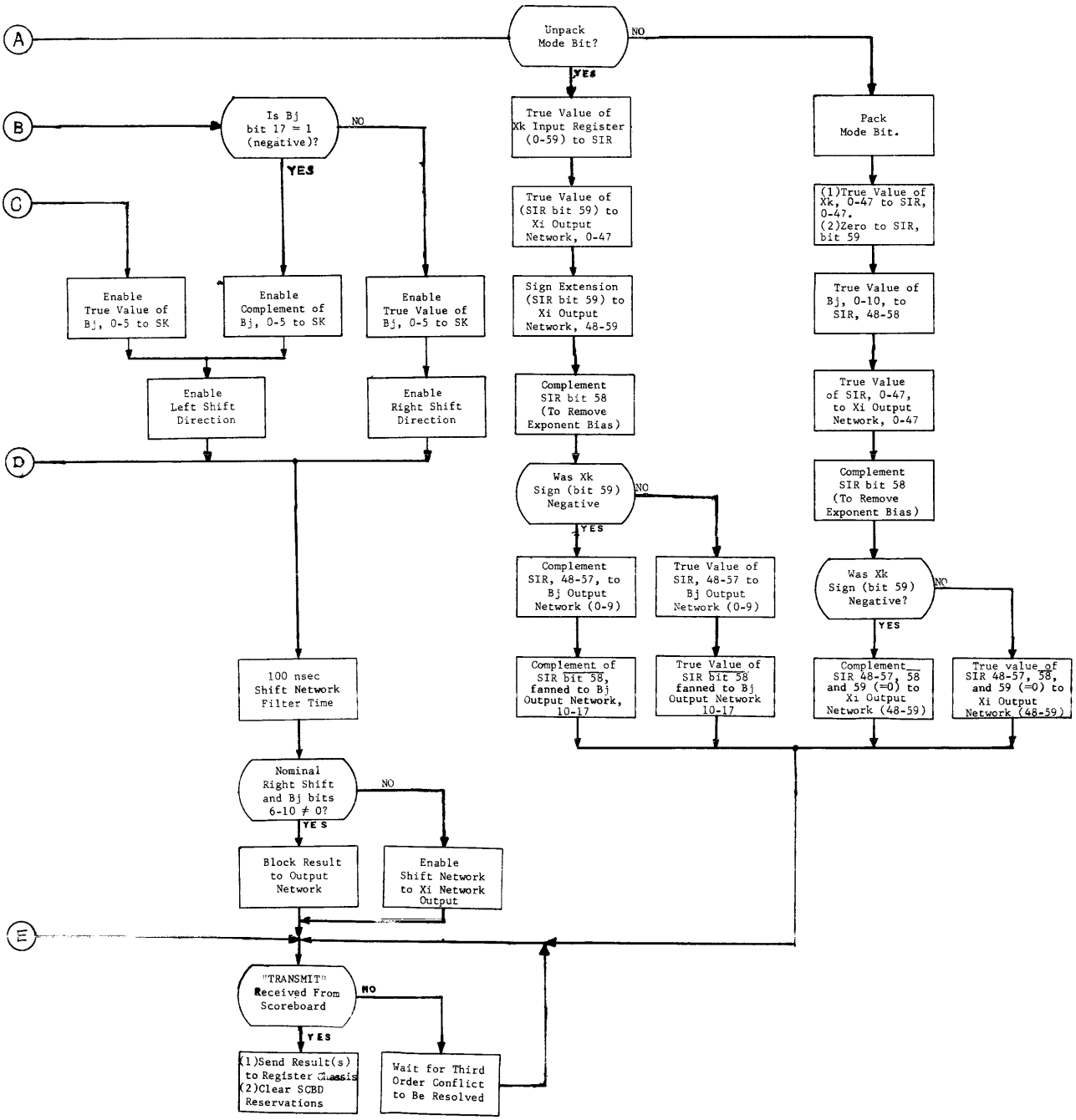


Figure 7.2-18B

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:

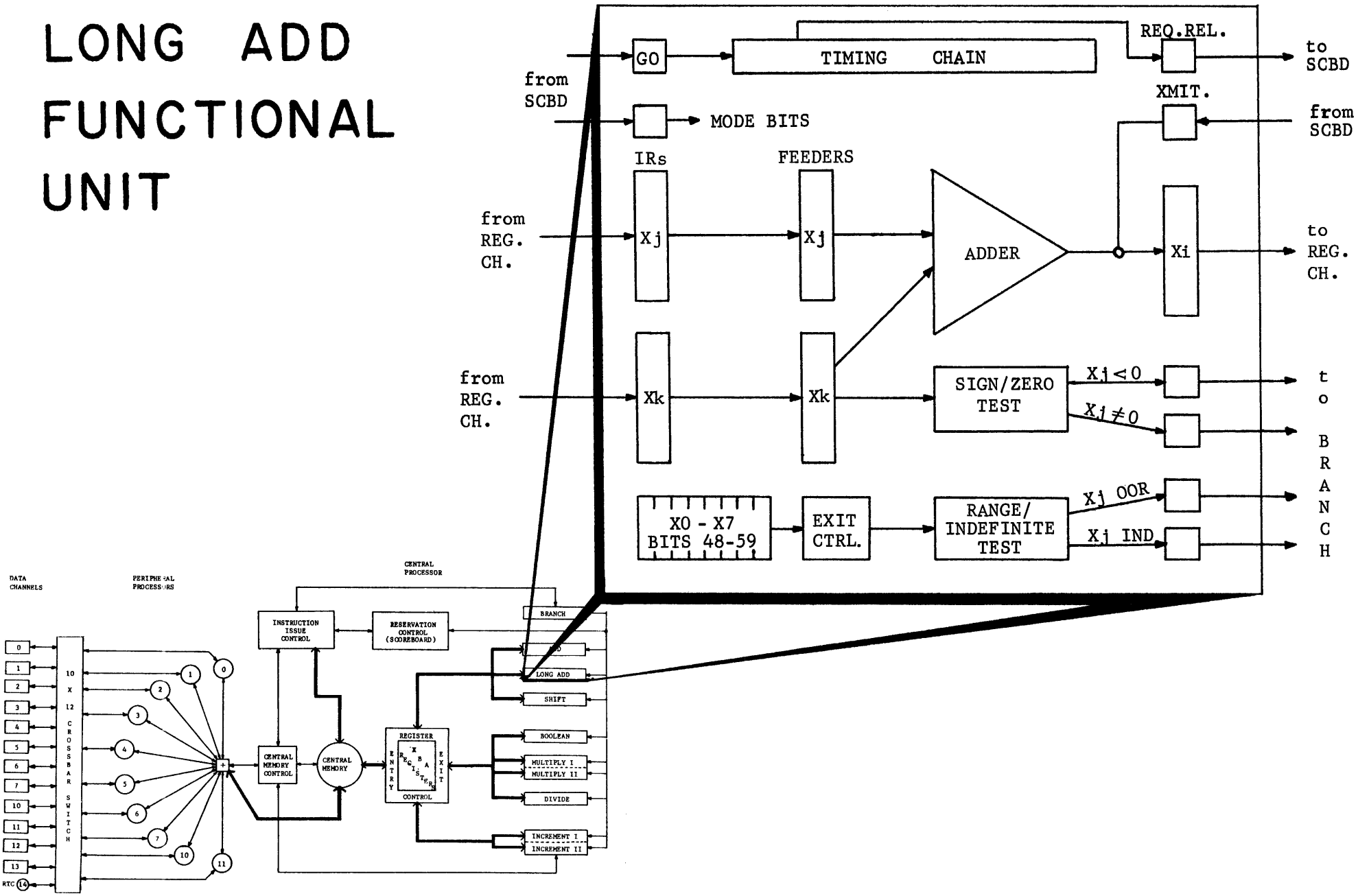
CONDITION ON KEs	RESULT
(EQUIVALENCE) (BORROW) $\Longrightarrow$	1
(EQUIVALENCE) ($\overline{\text{BORROW}}$) $\Longrightarrow$	0
($\overline{\text{EQUIVALENCE}}$) (BORROW) $\Longrightarrow$	0
($\overline{\text{EQUIVALENCE}}$) ($\overline{\text{BORROW}}$) $\Longrightarrow$	1

SECTION 7.3

LONG ADD

FUNCTIONAL UNIT

LONG ADD FUNCTIONAL UNIT



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 03X 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 <u>not</u> 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 03X Branch instructions, it is helpful to review the events that take place during the movement of any 0X instruction to the scoreboard. Recall that, in transferring a 0X 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 0X instructions none of the four Result flip/flops are set, hence the issuance of a 0X 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 03X 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 03X 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 Q_k 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 03X 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 (Xj and Xk) and two 3-bit tags (Fj and Fk) 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 Fk designator is set (setting Fj and Fi is disabled by the translation, fm = 0X). Also, setting RF1 and RF2 results in sending only the "Go Read" and F designator for Xk to Exit control. (The Fj designator, which equals zero, is sent to exit control, but the absence of the "Go Read Xj" inhibits translation; therefore, all zeros are sent on the Long Add Xj data trunk). In conclusion, both the j and k octals of U2 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 Xk" 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 fm = 0X translation, the Fi designator of the Long Add Unit is not set at scoreboard issue time. Fi will thus contain "0". Also, because of the 0X instruction, translation of Fi = 0 is inhibited (by the clear side "Full Bit" for Fi). When the "Request Release" arrives at the All Clear Network there is no Long Add Fi translation to compare with other Fj and Fk 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 Fi designator is cleared, a "Go Store" to Entry control is inhibited, although the three bit Fi designator (equal to zero) is sent. In order

to translate a code Fi = 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 Fj and Fi "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.

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 Xj and Xk to Xi (IXi = Xj + Xk)

Definition:

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

Data Flow:

The source operands, Xj and Xk are transferred from the chassis 8 input register to the feeder registers of the long add functional unit. Both Xj and Xk are transferred in true form. When a "transmit" signal is received from the scoreboard, the sum of Xj and Xk is gated to the result register Xi.

37 Integer Difference of Xj and Xk to Xi (IXi = Xj - Xk)

Definition:

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

Data Flow:

The source operands, Xj and Xk are transferred from the chassis 8

LONG ADD UNIT - BLOCK DIAGRAM

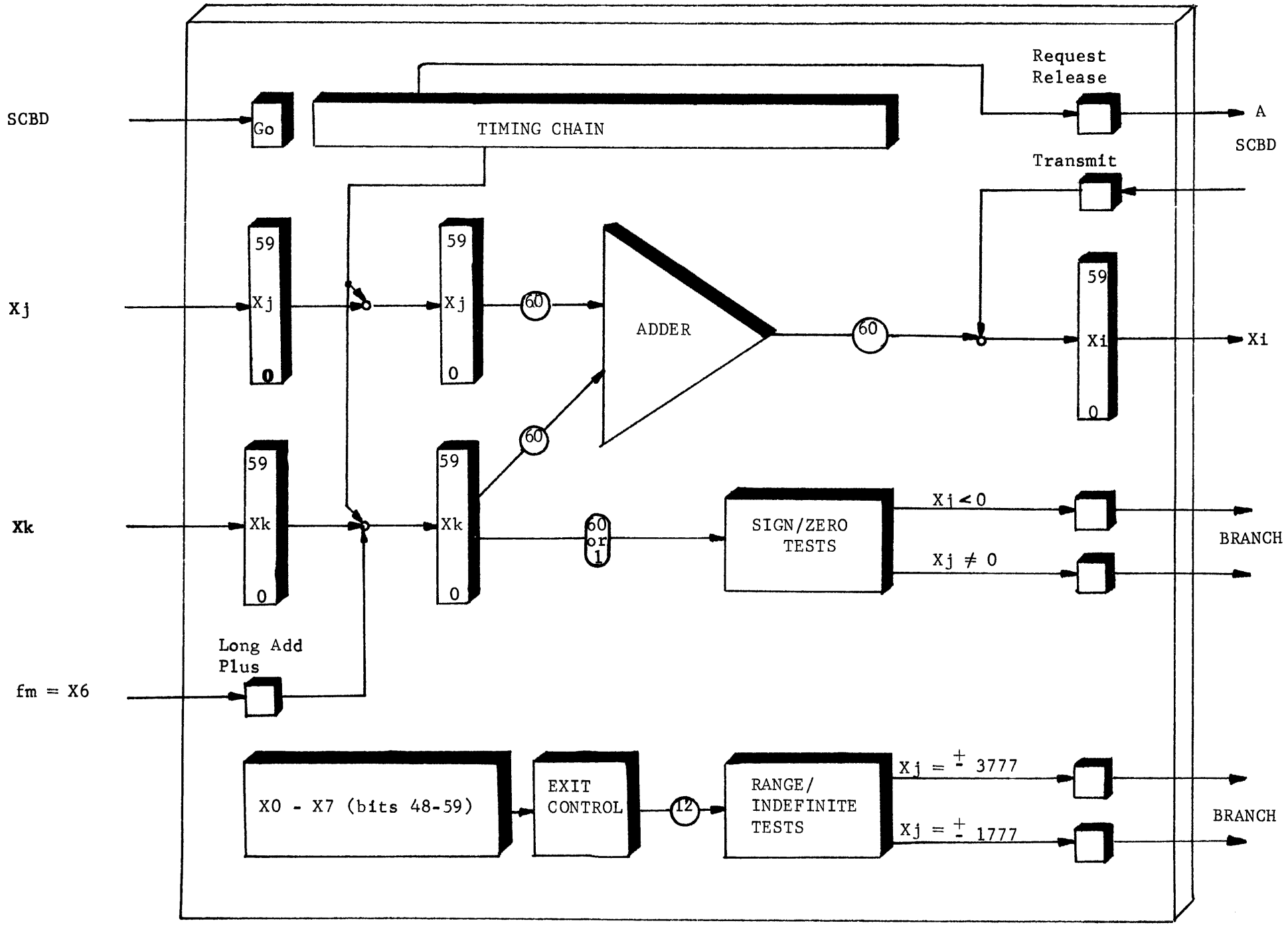


Figure 7.3-1

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

Conditional Branch Instructions

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

Definitions:

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

Test Validity:

030 & 031 - The zero tests check the full 60-bit word in Xj. 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 Xj. 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 Xj 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 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 Xj 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 Xk feeder register in complement form. There is no input to the Xj input register; it therefore contains all zeros. For sign tests, (32 + 33) a test of Xk bit 59 will indicate a negative or positive quantity. For the zero tests (030 + 031) all 60-bits of Xk 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 Xk 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 used by the Branch Unit to enable or disable the 03X conditional jump instructions.

---

* The computer will generate only positive indefinite results (1.e. 17770——0); however, the indefinite tests check for both positive and negative indefinite values.

7.3.3 MODE BIT

As far arithmetic processes are concerned, the Long Add unit is capable of performing only, addition and substraction. A single mode bit, called Long Add Plus is used to distinguish between these operations. It is generated with fm translation = X6 ANDed with Long Add Unit Busy. The true value of operand Xj is always sent to the Xj feeder register. During Addition, the Long Add Plus flip/flop is set enabling the true value of operand Xk to the Xk feeder register and forming the sum, Xi + Xk. During Substraction, the Long Add Plus flip/flop is cleared. Consequently, the complemented value of Xk is sent to the Xk feeder and the difference, Xj - Xk, is formed.

During the Branch (03X) instructions the Long Add Plus flip/flop is cleared. Therefore, the complemented value of the Xk input register is sent to the Xk 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.

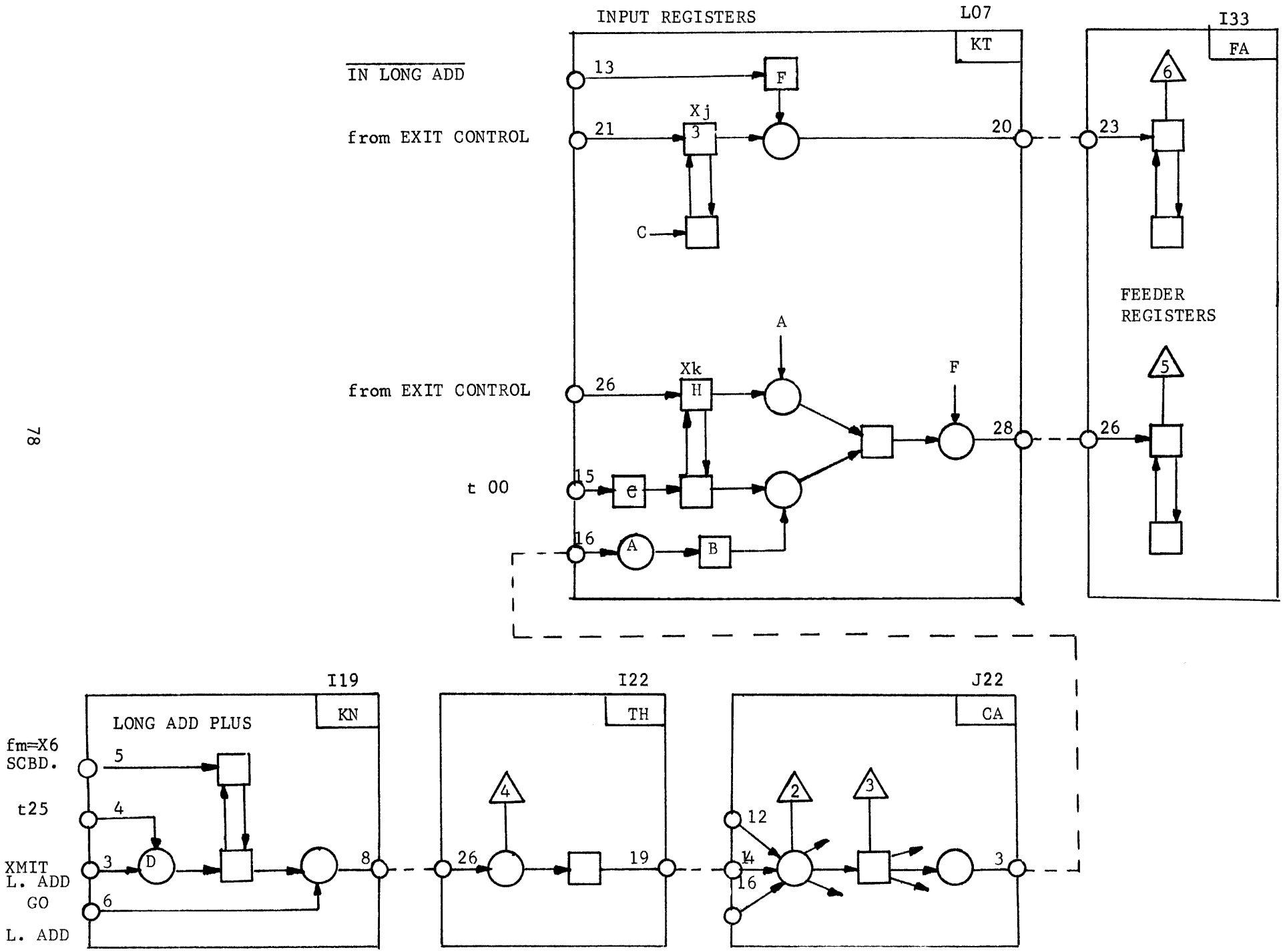
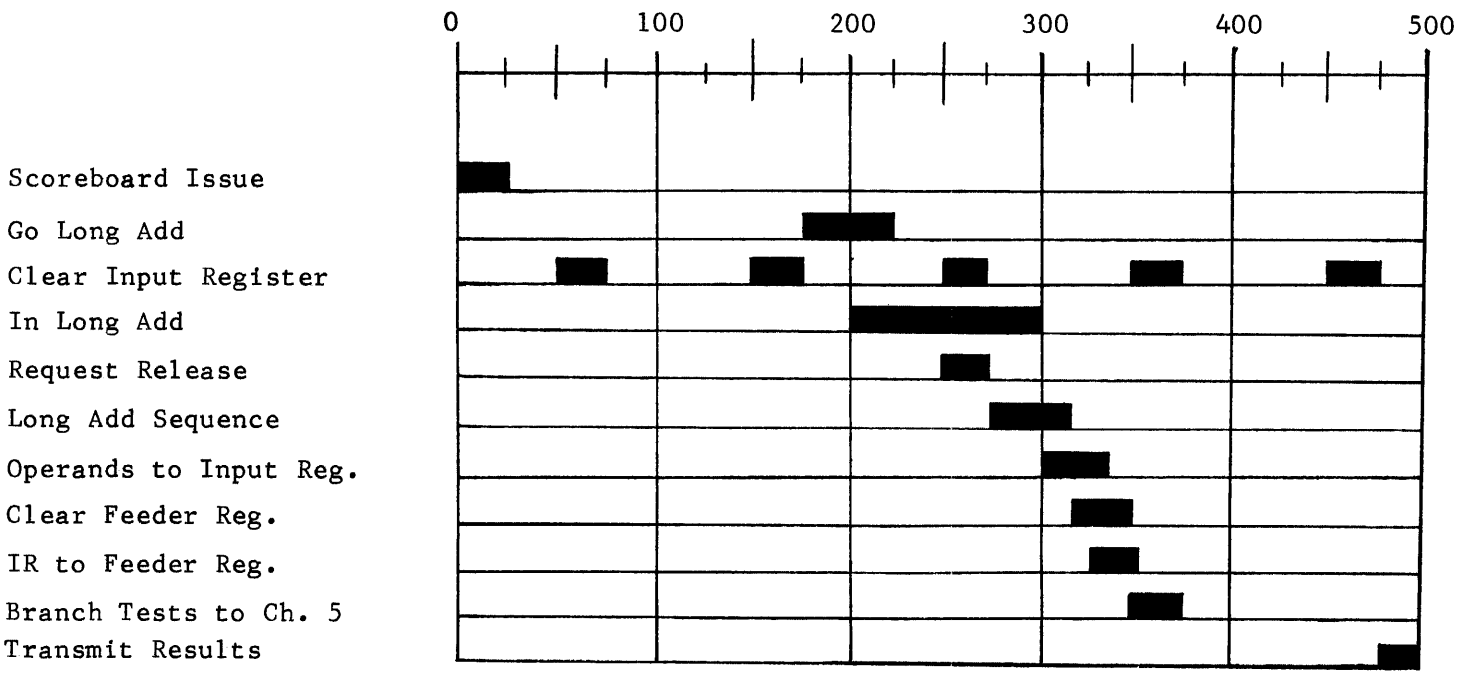

INPUT REGISTERS
L07
KT
I33
FA
IN LONG ADD
from EXIT CONTROL
13
F
Xj
21
3
20
23
6
C
FEEDER
REGISTERS
A
Xk
from EXIT CONTROL
26
H
F
28
26
5
t 00
15
e
16
A
B
I19
LONG ADD PLUS
KN
I22
TH
J22
CA
fm=X6
SCBD.
5
t25
4
XMIT
L. ADD
GO
3
D
8
6
L. ADD
26
19
4
12
14
16
2
3
3

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 base used (t000) 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 (8E28, TP 2) sets. Results in gating signal 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 (Xj + Xk) or difference (Xj - Xk) of the two source operands. To perform addition, both operands are sent to the feeders in true form. During subtraction, Xj is sent in true form and Xk is complemented. The decision to load the true or false value of Xk 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).

```
Additive        542---673
Adder           235---104
                777---777
```

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.

```
Subtractive     542---673    =    542---673
Adder          +235---105    =   -542---673
                                   000---000
```

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

	E	B	S	E
A =	0	0	1	1
B =	0	1	0	1

(A)

	E	B	S	E
A =	0	0	1	1
B =	1	0	1	0

(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.

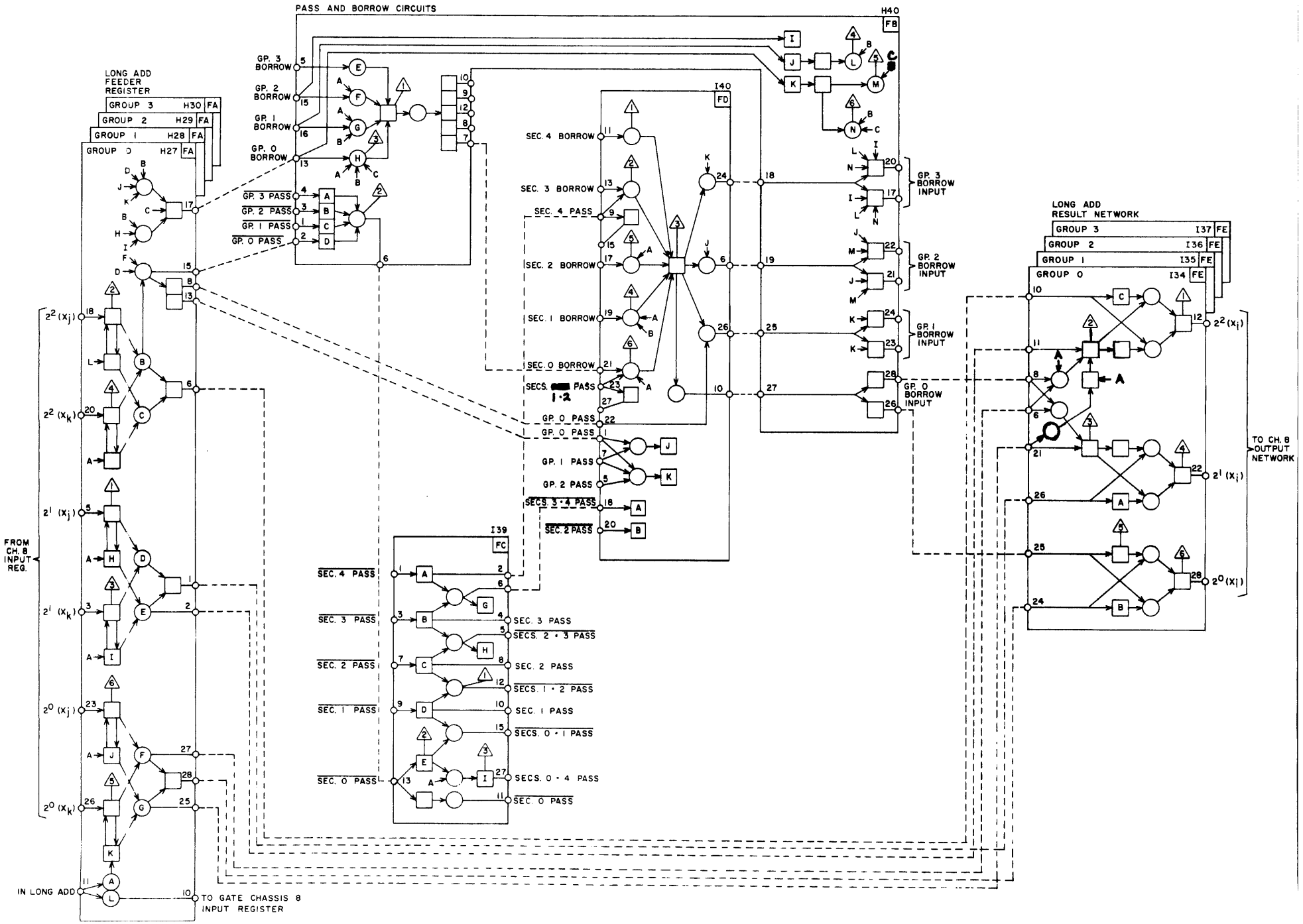

PASS AND BORROW CIRCUITS
H40
FB
I40
FD
I39
FC
LONG ADD FEEDER REGISTER
GROUP 3 H30 FA
GROUP 2 H29 FA
GROUP 1 H28 FA
GROUP 0 H27 FA
GP. 3 BORROW
GP. 2 BORROW
GP. 1 BORROW
GP. 0 BORROW
GP. 3 PASS
GP. 2 PASS
GP. 1 PASS
GP. 0 PASS
SEC. 4 BORROW
SEC. 3 BORROW
SEC. 4 PASS
SEC. 2 BORROW
SEC. 1 BORROW
SEC. 0 BORROW
SECS. 1·2 PASS
GP. 0 PASS
GP. 0 PASS
GP. 1 PASS
GP. 2 PASS
SECS. 3·4 PASS
SEC. 2 PASS
GP. 3 BORROW INPUT
GP. 2 BORROW INPUT
GP. 1 BORROW INPUT
GP. 0 BORROW INPUT
LONG ADD RESULT NETWORK
GROUP 3 I37 FE
GROUP 2 I36 FE
GROUP 1 I35 FE
GROUP 0 I34 FE
TO CH. 8 OUTPUT NETWORK
FROM CH. 8 INPUT REG.
IN LONG ADD
TO GATE CHASSIS 8 INPUT REGISTER
SEC. 4 PASS
SEC. 3 PASS
SEC. 2 PASS
SEC. 1 PASS
SEC. 0 PASS
SEC. 3 PASS
SECS. 2 • 3 PASS
SEC. 2 PASS
SECS. 1 • 2 PASS
SEC. 1 PASS
SECS. 0 • 1 PASS
SECS. 0 • 4 PASS
SEC. 0 PASS

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 25 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. For instance, test Point 1 on the FB modules, 1) checks for borrows generated in each of groups 0, 1, 2 and 3, 2) checks for satisfies ($\overline{\text{Passes}}$) in the groups and 3) 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, ($\overline{\text{Satifies}}$) 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 2, 3, and 4 contained no satisfies) OR (Section 0 generated a borrow and Sections 1, 2, 3 and 4 contained no satisfies)) on the right half of the FB modules, borrows and passes for group 0, 1, 2, and 3 are combined to determine 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 od the FE modules.

FE Modules

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

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

$$(\text{Enable})(\underline{\text{Borrow}}) \Rightarrow 1$$

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

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 "one"; with a borrow the result should be "zero". In Boolean -

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

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

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.

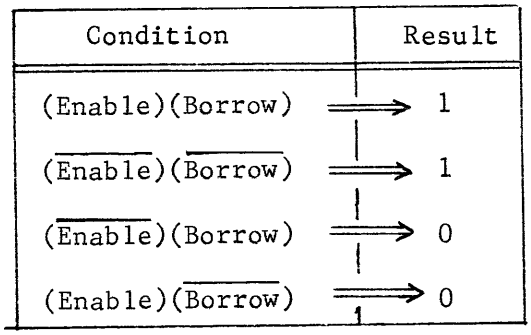

Condition
Result
(Enable)(Borrow)
1
(Enable)(Borrow)
1
(Enable)(Borrow)
0
(Enable)(Borrow)
0

Figure 7.3-6

7.3.6 BRANCH TESTS

The Branch tests are used to condition the 03X series of jump intructions. Four tests (Zero, Sign, Range, and Indefinite) are used as follows:

Opcode	Name	Test
030	Jump to K if Xj = 0	zero
031	Jump to K if Xj ≠ 0	zero
032	Jump to K if Xj ≥ 0	sign
033	Jump to K if Xj < 0	sign
034	Jump to K if Xj is in Range	range
035	Jump to K if Xj is out of Range	range
036	Jump to K if Xj is definite	indefinite
037	Jump to K if Xj is indefinite	indefinite

Zero Test

The zero test circuitry tests all 60-bits of the Xk 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 Xk input register is sent to the Xk feeder. The Xj input register receives no input; consequently, the Xj feeder contains all zeros.

For the positive zero test, a check for non equivalence between each bit of Xj and Xk 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 $Xj \neq 0$ signal to the Branch unit. Conversely, if all bit positions are non-equivalent, the transmitter is disabled - the absence of the $Xj \neq 0$ signal implies $Xj = 0$.

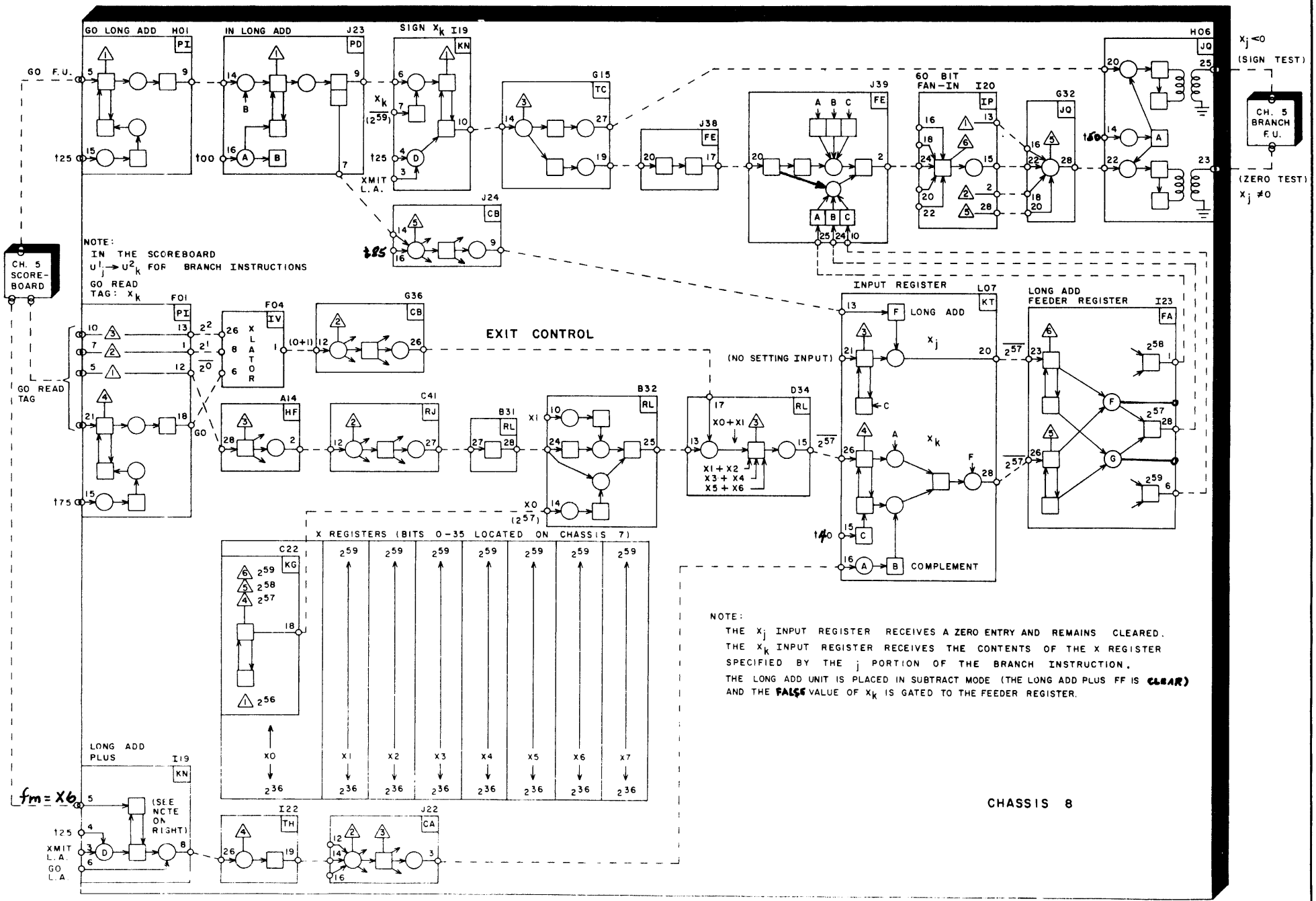
The Negative Zero test works in a similar manner. A check is made for all bit positions being equivalent. Any position resulting in non-equivalence will enable the $Xj \neq 0$ signal to be transmitted. The absence of the $Xj \neq 0$ signal will indicate to the branch unit that $Xj = 0$.

Sign Test

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

Range Test

The range test checks bits 48-59 of Xj 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.



SIGN/ZERO TESTS
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 ($1XX\ XXX\ 000\ 000_2$)

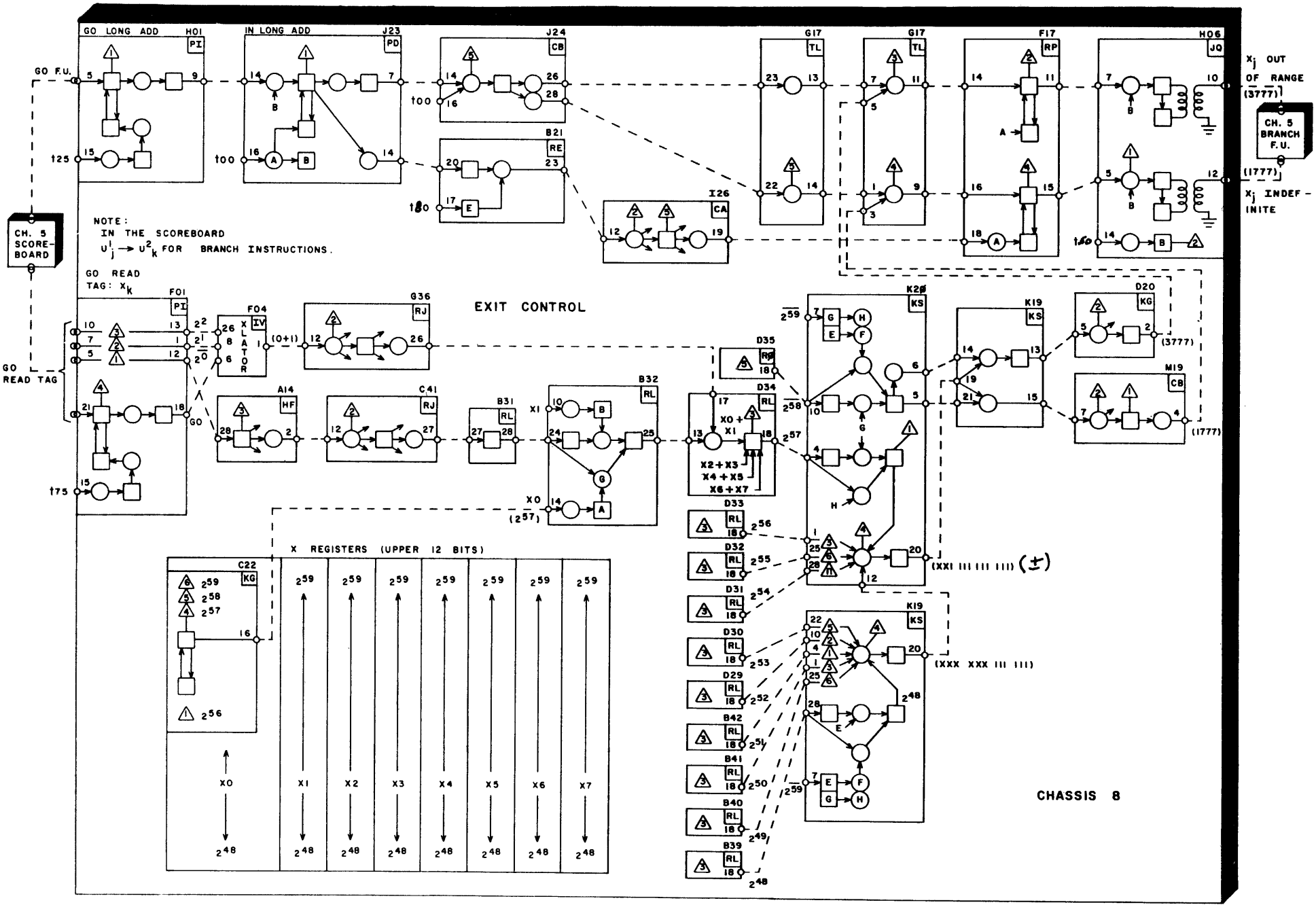
The check is made by comparing each bit position with the sign (bit 59) of the register (terms E and F). K19, pin 20 then feeds pin 12 of K20, the second KS module, along with bits 54, 55, and 56. At test point 4, all the bit states are combined and the output of pin 20 indicates:

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 $\pm 1777_8$ (indefinite) or $\pm 3777_8$ (out of range) or neither. The translation for K19, pin 14 ($(58)(\overline{59})\ v\ (\overline{58})(59)$) 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.

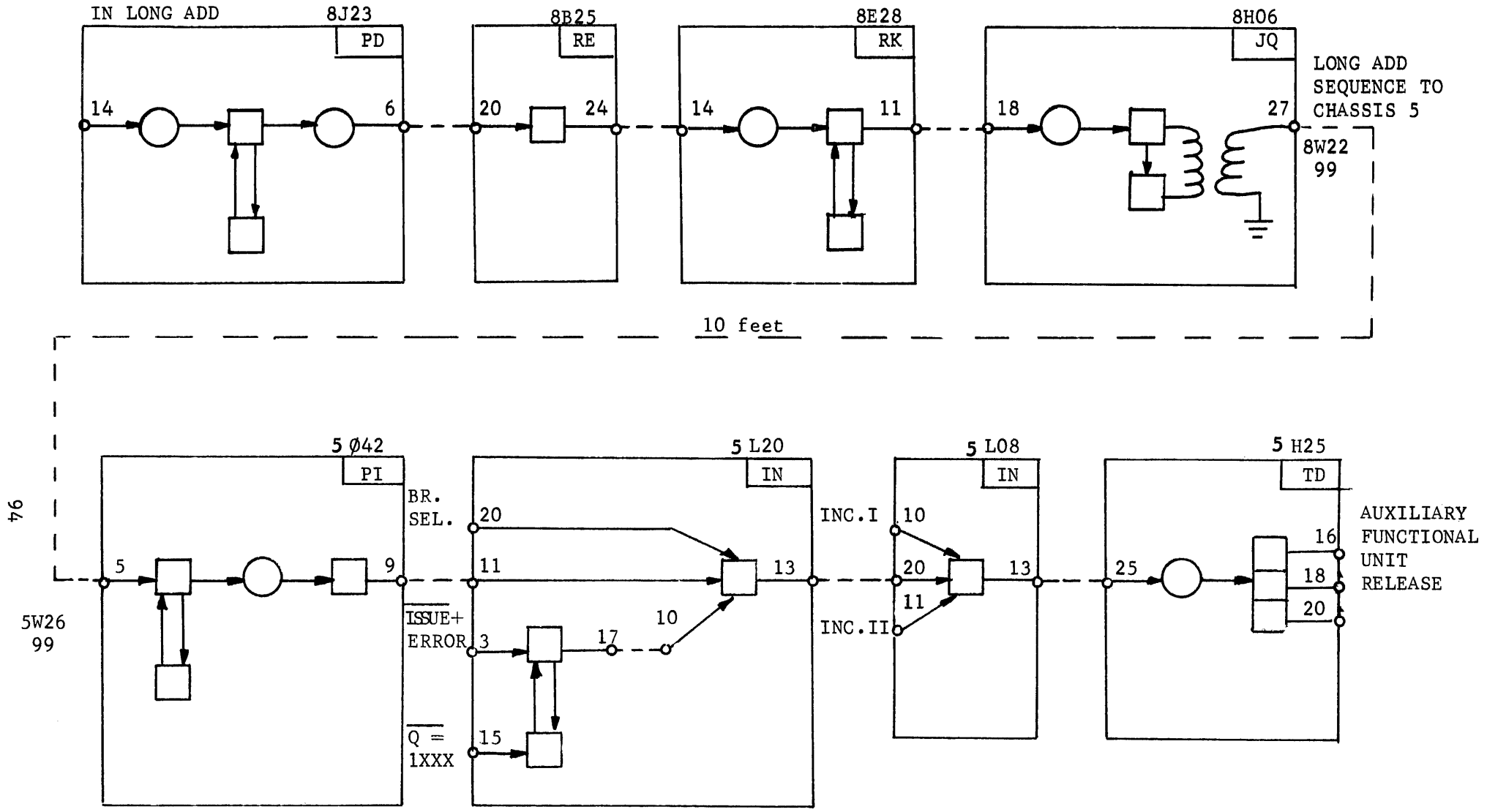


RANGE/ INDEFINITE TESTS
Figure 7.3-8

Indefinite Test

This test utilizes the same circuitry as the range test (Figure 7.3-8) except that the K19, pin 19 translation ($0X1\ 111\ 111\ 111_2$ or $1X0\ 000\ 000\ 000_2$) is ANDed with K19, pin 21. When equal to a "L" this pin indecates $(\overline{58})(\overline{59})$ 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" signal to the Branch unit.

Synchronization of the Long Add Unit Branch test results 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 signal called "Auxiliary 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)



AUXILLIARY FUNCTIONAL UNIT RELEASE

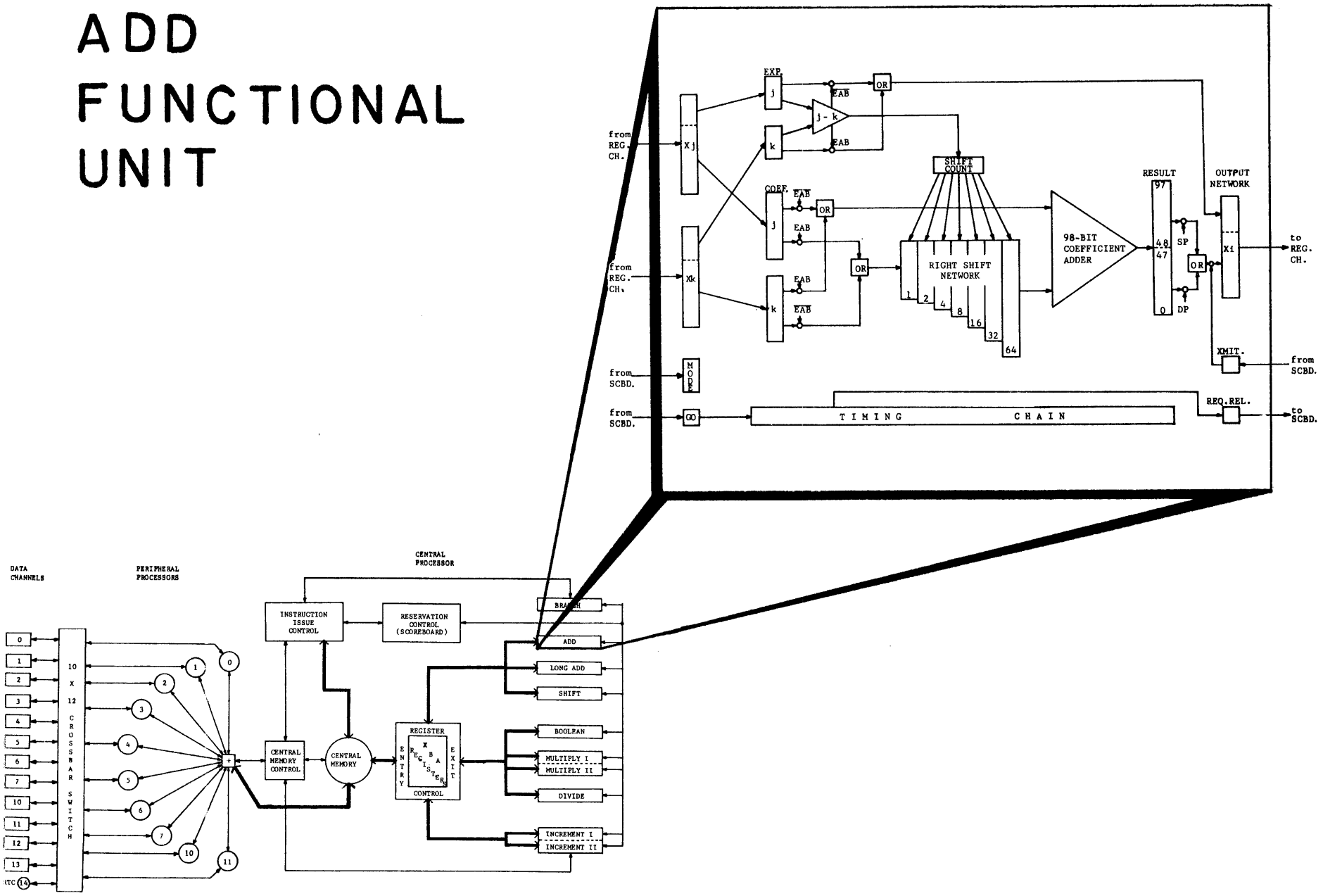
Figure 7.3-9

SECTION 7.4

ADD

FUNCTIONAL UNIT

ADD FUNCTIONAL UNIT



ADD FUNCTIONAL UNIT

7.4.1 INTRODUCTION

The Add Functional Unit is utilized to perform floating point addition and subtraction of 60 bit operands. The computations may be made in rounded single precision or unrounded single and double precision. In any case, the unit cycle time is 400 nanoseconds.

The following instructions (discussed in detail in section 7.4.2) use the Add Unit.

30 Floating SUM of Xj and Xk to Xi

31 Floating DIFFERENCE of Xj and Xk to Xi

32 Floating D.P. SUM of Xj and Xk to Xi

33 Floating D.P. DIFFERENCE of Xj and Xk to Xi

34 ROUND Floating SUM of Xj and Xk to Xi

35 ROUND Floating DIFFERENCE of Xj and Xk to Xi

The Add unit is located on data trunk number 1 along with the Shift and Long Add Units. It holds first priority in reading operands and second priority for storing results.

The remainder of this discussion assumes a knowledge of the 6000 Series floating point. If a review is desired, refer to Appendix A.

Before addition of two floating point numbers takes place, the exponents must be equalized. This is accomplished in the Add Unit by subtracting the smaller exponent from the larger and using this

difference to enable a Right Shift Network. The coefficient with the smaller exponent is right shifted before being fed to the adder. The coefficient with the larger exponent is fed directly to the adder feeder registers. The larger exponent will be the final exponent of the result, (it may be adjusted for overflow, or double precision results).

The Add Unit always generates a 96 - bit result which is the sum or difference of two 48 bit coefficients (Xj and Xk) If single precision is selected, the upper 48 - bits of the 96 - bit result and the larger exponent Xj or Xk is returned to Xi. Selecting double precision causes the lower 48 bits of the 96 bit result and the larger exponent minus $60_{(8)}$ to be returned to Xi. $60_{(8)}$ is subtracted since selection of the lower 48 - bits during double precision effectively moves the binary point $48_{(10)}$ placed to the right, there by increasing the coefficient magnitude. To compensate, the exponent must be decremented. The following example illustrates the addition of two quantities in single and double precision.

ADD: Xj = 2005 0 ——— 05244 (5244.2^5)

Xk = 2016 0 ——— 07305 (7305.2^{16})

1) The difference of the exponents is $11_{(8)}$. This implies shifting the coefficient with the smaller exponent (Xj) right $11_{(8)}$ places.

Xk 0 —— 07305.0 ———— 0

Xj (RS $11_{(8)}$) 0 ——— 05.2440 ———— 0

2) 96 bit result = $\underbrace{0 \text{ —— } 07312}_{\text{upper}} \cdot \underbrace{2440 \text{ ——— } 0}_{\text{lower}}$

3) Single Precision Result: 2016 $\underbrace{0 \text{ —— } 07312}_{\text{upper}}$

4) Double Precision Result: 1735 $\underbrace{2440 \text{ ——— } 0}_{\text{lower}}$

(EXP. = 1735 = 2016 - 60)

A coefficient overflow condition is corrected by right shifting the 96-bit coefficient one place and incrementing the exponent by one. Should this result in exponent overflow (3777) the right shifted coefficient and an exponent of 3777 will be returned to the result register. For example:

377640 ——— 0 + 3776520 —— 0 = 3777450 —— 0

In the event that either Xj or Xk is initially infinite (3777 or 4000) an exponent of 3777 and an all zero coefficient is returned to Xi. For example:

37770 ——— 0 + 3050 720 —— 0 = 37770 ——— 0

7.4.2 INSTRUCTION LIST / DATA FLOW

The following instructions will select the ADD functional unit. The terms in parenthesis following the instruction name are the ASCENT symbolic codes used in assembler coding. Data flow can be followed on figure 7.4.1.

30 FLOATING SUM of Xj and Xk to Xi (FXi = Xj + Xk)

DEFINITION:

This instruction forms the sum of the floating point quantities from operand registers Xj and Xk and packs the result in operand register Xi. The packed result is the upper half of a double precision sum. After the functional receives the operands the following occurs. At the start both arguments are unpacked, and the coefficient of the argument with the smaller exponent is entered into the upper half of a 98-bit accumulator. The coefficient is shifted right by the difference of the exponents. The other coefficient is then added into the upper half of the accumulator. If overflow occurs, the sum is right-shifted one place and the exponent of the result increased by one. The upper half of the accumulator holds the coefficient of the sum, which is not necessarily in normalized form. The exponent and upper coefficient are then repacked in operand register Xi.

If both exponents are zero and no overflow occurs, the instruction effects an ordinary integer addition.

DATA FLOW: The true value of Xj and Xk are entered into the exponent and coefficient feeders. The exponents are subtracted and the coefficient of the smaller exponent is gated through the right shift

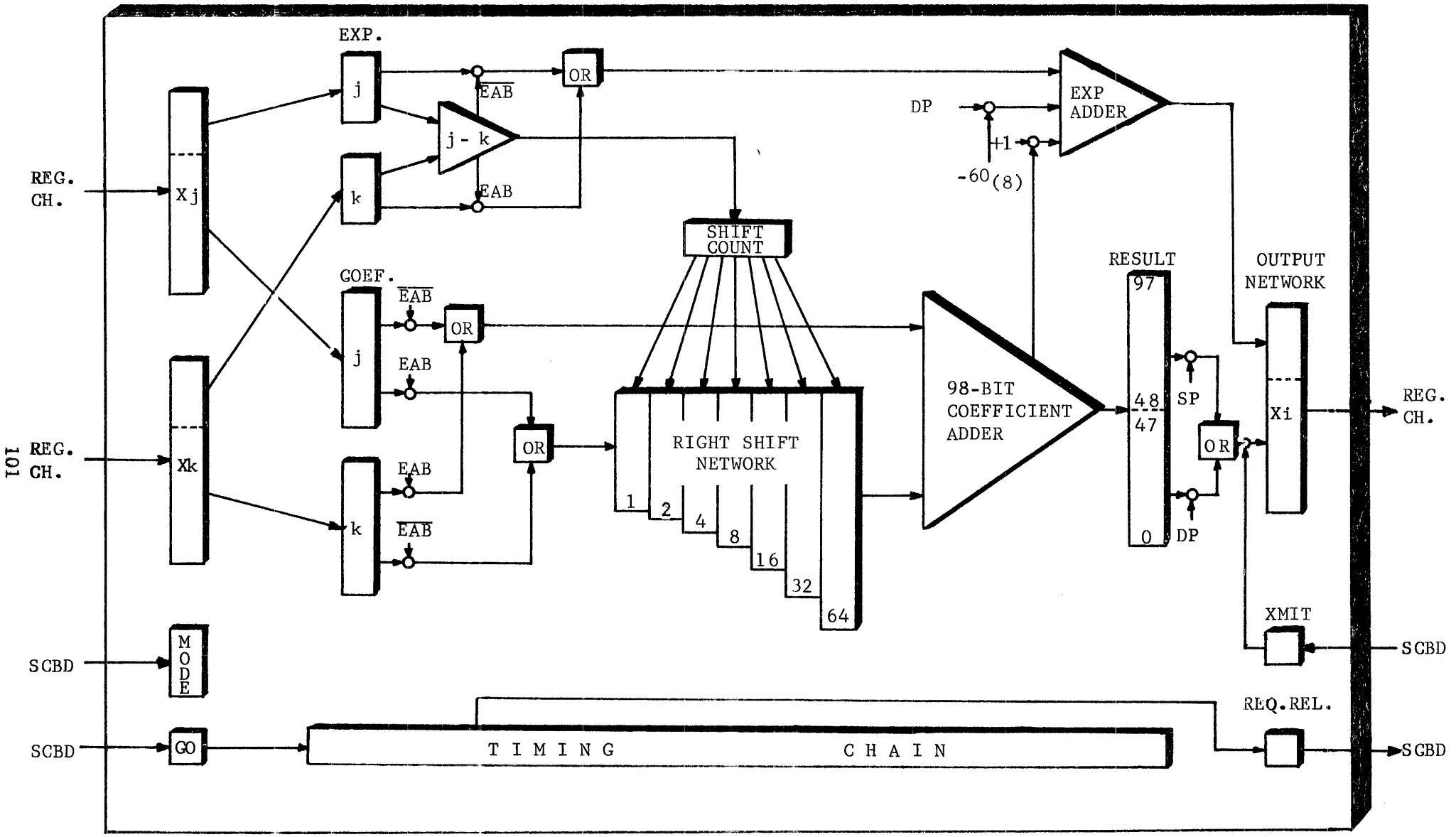


Figure 7.4-1

ADD FUNCTIONAL UNIT

network. It is shifted the number of places specified by the absolute value of the difference of exponents. The shifted value feeds the 98-bit adder along with the unshifted coefficient. If coefficient overflow occurs, (i.e.2^{96} of result = 1) the coefficient is right shifted one place and the final exponent incremented by one. The upper 48-bits of the 96-bit sum are gated to bits 0-47 of the output network. The final exponent will be the larger of the two original exponents (Possibly incremented due to overflow).

31. FLOATING DIFFERENCE of Xj and Xk to Xi (Xi = Xj - Xk)

DEFINITION:

This instruction forms the difference of the floating point quantities from operand registers Xj and packs result in operand register Xi. Alignment and overflow operations are similar to the Floating Sum (30) instruction, and the difference is not necessarily normalized. The packed result is the upper half of a double precision difference.

An ordinary integer subtraction is performed when the exponents are zero.

DATA FLOW: Data flow is the same as for the 30 instruction with the following exception. The coefficient of the operand Xk is complemented into the feeder register and hence to the adder. The adder forms the sum of the true value Xj and complemented Xk, thereby generating the difference, Xj - Xk.

32 FLOATING DOUBLE PRECISION SUM of Xj and Xk to Xi

(DXi = Xj + Xk)

DEFINITION:

This instruction forms the sum of two floating point numbers as in the 30 instruction, but packs the lower half of the double precision sum with an exponent 48 less than the upper sum.

DATA FLOW: Data flow is the same as the 30 instruction with the following exceptions. 1) Bits 0 - 47 of the 96-bit sum are sent to the output network. 2) $60_{(8)}$ is subtracted from the larger of the two original exponents to compensate for selection of the lower sum. The final exponent may therefore be the larger exponent minus $60_{(8)}$ or minus $57_{(8)}$ (in the event that coefficient overflow was encountered and a right shift one place was required).

33 FLOATING DOUBLE PRECISION DIFFERENCE of Xj and Xk to Xi

(DXi = Xj - Xk)

DEFINITION:

This instruction forms the difference of two floating point numbers as in the Floating Difference (31) instruction, but packs the lower half of the double precision difference with an exponent of 48 less than the upper sum.

DATA FLOW: Data flow is the same as for the 32 instruction with the following exception. The coefficient of the operand Xk is complemented into the feeder register and hence to the adder. The adder forms the sum of the true value Xj and complemented Xk thereby

generating the difference, Xj - Xk.

34 ROUND FLOATING SUM of Xj and Xk to Xi (RXi = Xj + Xk)

DEFINITION:

This instruction forms the round sum of the floating point quantities from operand registers Xj and Xk and packs the upper sum of the double precision result in operand register Xi. The sum is formed in the same manner as the Floating Sum instruction but the operands are rounded before the addition, as shown below, to produce a round sum.

1) A round bit is attached at the right end of both operands if:
   a) both operands are normalized, or
   b) the operands have unlike signs.

2) A round bit is attached at the right endo of the operand with the larger exponent for all other cases.

DATA FLOW: The data flow is the same as for the 30 opcode with the exception of attaching round bits prior to adding the coefficient. Round bits are entered into the feeder registers under the conditions stated in the above definition. They cause carries to be propagated into the significant (upper) half of the coefficient if the lower half equals $\frac{1}{2}$ or greater, thus performing a $\frac{1}{2}$ round.

35 ROUND FLOATING DIFFERENCE of Xj and Xk to Xi

(RXi = Xj - Xk)

DEFINITION:

This instruction forms the round difference of the floating point quantities from operand registers Xj and Xk and packs the upper difference of the double precision result in operand register Xi. The difference is formed in the same manner as the Floating Difference (31) instruction but the operands are rounded before the subtraction, as shown below, to produce a round difference.

1) A round bit is attached at the right end of both operands if:
   a) both operands are normalized, or
   b) the operands have like signs.
2) A round bit is attached at the right end of the operand with the larger exponent for all other cases.

DATA FLOW: Data flow is the same as for the 35 opcode with the following exceptions: 1) The Xk coefficient is complemented into the feeders so that the difference may be formed by addition. 2) The round bits are attached for the conditions stated in the definition to cause carry propagation in a manner which will cause a $\frac{1}{2}$ round to occur during difference operations.

7.4.3 MODE BITS

Three mode bits are used by the Add unit to distinguish between the six floating add opcodes. The mode bit names and corresponding opcodes are:

ADD PLUS	30,32,34
ROUND ADD	34,35
DOUBLE PRECISION	32,33

Figure 7.4-2 and the chassis 8 wire tabs should be referenced during the following discussion.

ADD PLUS - The Add Plus mode bit (G01, TP6) is generated by single precision, double precision and round add opcodes. It allows the true value of the coefficient of operand Xk to be gated to the adder feeder when a sum operation is to be performed. The absence of the Add Plus mode bit causes the Xk coefficient to be complemented into the feeder. The subsequent addition causes Xk to be subtracted from Xj (by addition of the complement). The actual transfer into the adder feeders occurs after the receipt of the "Go Add" signal (M29 pin 23). Section 7.4.4 gives the detailed timing analysis.

ROUND ADD - The "Round" mode bit is generated by opcodes 34 & 35 (rounded sum and difference). The bit is caught on 8H01, TP4 (Figure 7.4-2) which sets E19, TP2 which remains set until the "Transmit" signal is received. Pin 9 is fanned out on H20 to two AND gates (H09 and J02) whose outputs are ORed on module Ø32. Pin 13 of Ø32 translates as:

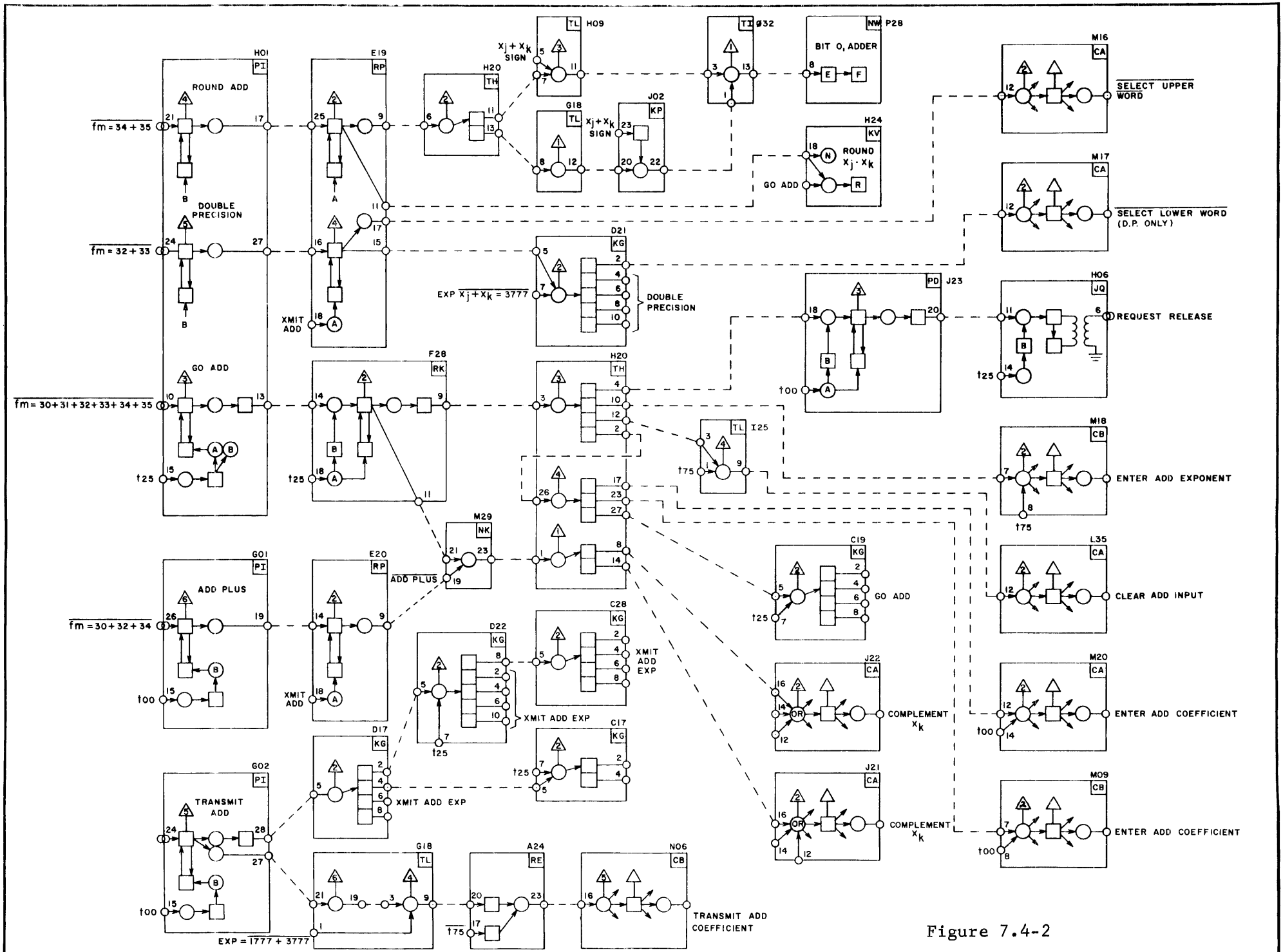

H01 PI
ROUND ADD
fm = 34 + 35
DOUBLE PRECISION
fm = 32 + 33
GO ADD
fm = 30+31+32+33+34+35
t25
G01 PI
ADD PLUS
fm = 30+32+34
t00
G02 PI
TRANSMIT ADD
t00
EXP = 1777 + 3777
E19 RP
XMIT ADD
F28 RK
t25
M29 NK
ADD PLUS
E20 RP
XMIT ADD
H20 TH
H09 TL
Xj + Xk SIGN
G18 TL
Xj + Xk SIGN
J02 KP
T032
NW P28
BIT 0, ADDER
E F
H24 KV
ROUND Xj · Xk
GO ADD
R
N
D21 KG
EXP Xj+Xk = 3777
DOUBLE PRECISION
H20 TH
I25 TL
t75
D22 KG
XMIT ADD EXP
C28 KG
XMIT ADD EXP
t25
D17 KG
XMIT ADD EXP
C17 KG
t25
G18 TL
t75
A24 RE
N06 CB
TRANSMIT ADD COEFFICIENT
PD J23
t00
H06 JQ
REQUEST RELEASE
t25
C19 KG
GO ADD
t25
J22 CA
COMPLEMENT Xk
J21 CA
COMPLEMENT Xk
M16 CA
SELECT UPPER WORD
M17 CA
SELECT LOWER WORD (D.P. ONLY)
M18 CB
ENTER ADD EXPONENT
t75
L35 CA
CLEAR ADD INPUT
M20 CA
ENTER ADD COEFFICIENT
t00
M09 CB
ENTER ADD COEFFICIENT
t00

Figure 7.4-2

(Round)(Positive) + ($\overline{\text{Round}}$)(Negative)

The negative and positive terms refer to the sign of the coefficient with the larger exponent; i.e. the unshifted coefficient. If it is positive, and "Round" is specified, a one is placed to the right (bit 2^{47}) of the unshifted coefficient. This in essence adds $\frac{1}{2}$ to the coefficient with the larger exponent. If bit 47 of the second operand (from the shift network) is a "1" (i.e. bits 0 - 47 $\geq \frac{1}{2}$) a carry into bit 48 will be generated during addition. Thus, a $\frac{1}{2}$ round has been performed. If the "Round" mode is not specified but the unshifted operand is negative it is necessary to set bit 2^{47} to prevent the Round from occurring. This is justified by recalling that negative coefficients enter the feeders in complement form. Thus, when rounding a negative (complemented) number, bit 2^{47} should be a "0"; to disable rounding, a "1".

Round bits may also be attached below the shifted operand in certain cases. Decoding pins 14 and 19 of 8H24 (C.E. Diagrams, Sheet 223) yields the following formulas:

$$\text{pin 14} \Longrightarrow \left[(\overline{RD})(-Xj)\right] + \left[RD\right]\left[(+Xj)(-Xk) + (+Xj)(Xj2^{47})(Xk2^{47}) + (-Xj)(-Xk)(Xj2^{47} + Xk2^{47})\right] \Longrightarrow \text{ROUND Xj.}$$

$$\text{pin 19} \Longrightarrow \left[(\overline{RD})(-Xk)\right] + \left[RD\right]\left[(+Xk)(-Xk) + (+Xk)(Xj2^{47})(Xk2^{47}) + (-Xk)(-Xj)(Xj2^{47} + Xk2^{47})\right] \Longrightarrow \text{ROUND Xk}$$

These conditions are stored in flip-flops on module K21 (sheet 223). Determination of whether Xj or Xk is sent through the shift network is made with terms A (shift Xk) and D (shift Xj) on K21.

The operand which has the smaller exponent will be entered into the shift network and will be rounded by entering a round bit in position 2^{47}. As the operand filters through the shift network the round bit is shifted along with the operand itself.

DOUBLE PRECISION - The D.P. mode bit is received on 8H01, TP3 from chassis 5 (Figure 7.4-2). The bit is ANDed with the condition "Xj • Xk ≠ 3777" and is fanned out on module D21. Pin 2 is tied to module M17, another fanout, which enables selection of the lower 48 bits of the 96 - bit sum. Pins 4, 6, 8 and 10 of D21 go to the exponent adder which subtracts $60_{(8)}$ from the larger exponent; these pins force the value, $-60_{(8)}$ into the adder during double precision operations. (See Section 7.4.5 for the exponent adder logic analysis).

7.4.4 TIMING SEQUENCE

The timing diagram for the Add Functional unit is shown in Figure 7.4-3 and should be referenced during the following discussion. The C.E. Diagrams and Wire Tabs should also be used if proof of the timing sequence is desired. Each term on the timing diagram is given a number which is used to sequence this explanation. The scoreboard issue for an Add opcode occurs at t-100 (not shown in Figure 7.4-3)

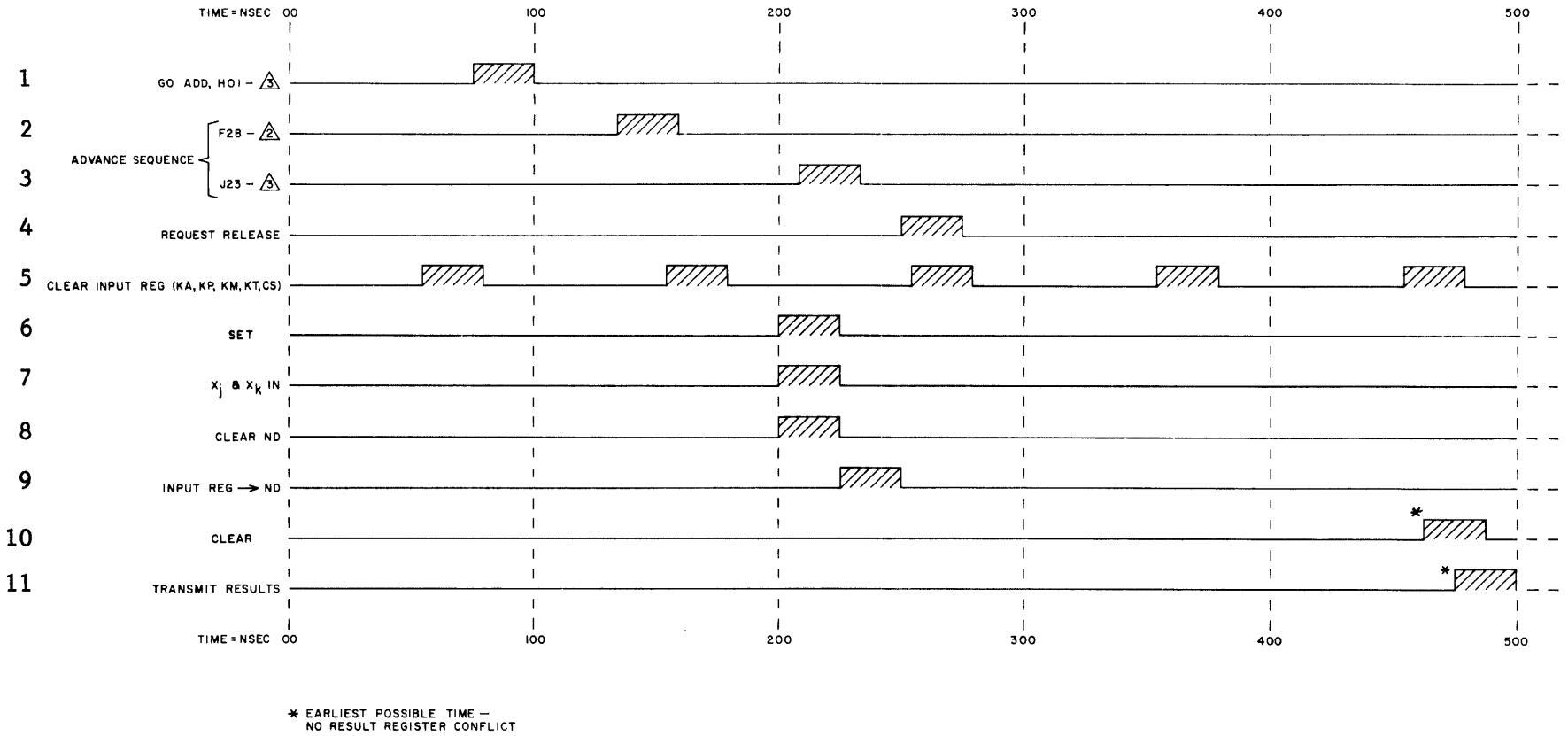

TIME = NSEC 00
100
200
300
400
500
1
GO ADD, HOI – 3
2
ADVANCE SEQUENCE
F28 – 2
3
J23 – 3
4
REQUEST RELEASE
5
CLEAR INPUT REG (KA, KP, KM, KT, CS)
6
SET
7
X_j & X_k IN
8
CLEAR ND
9
INPUT REG → ND
10
CLEAR
11
TRANSMIT RESULTS
TIME = NSEC 00
100
200
300
400
500
* EARLIEST POSSIBLE TIME – NO RESULT REGISTER CONFLICT

Figure 7.4-3

1) GO ADD - This term indicates the time that the "Go Add" signal is received on chassis 8. The signal starts the Add timing chain (flip-flops) which sequences the Add operation.

2) & 3) ADVANCE SEQUENCE - These terms show the setting of the two flip-flops of the Add timing chain (Refer to Figure 7.5-2). These flip-flops sequence events of the Add operation.

4) REQUEST RELEASE - This signal is sent to the Scoreboard at about t250. The Scoreboard checks for the presence of third order conflicts; if none exists, the "Release" condition generates a "Transmit" which is received on chassis 8 about 175 nanoseconds later (See term #11).

5) CLEAR INPUT REGISTERS - The chassis 8 input registers are cleared every minor cycle to enable the receipt of operands on chassis #8 every 100 nanoseconds (i.e. 1 megacycle data trunk rate).

6) SET NM - This term shows the setting of the exponent feeder registers, on chassis 8. They are set from the portion of Exit Control located on chassis 8 and do not arrive via receiver modules as do bits 0 - 35.

7) Xj & Xk IN - This term indicates the time that bits 0-35 of Xj and Xk are received from chassis 7. The input (catching) register is composed of K A modules which also serve to gate operands to the Long Add and Shift units.

8) CLEAR ND - The ND modules are the feeder registers for the coefficient adder. At this point they are cleared in preparation for the transfer of the operands from the input registers.

9) INPUT REG $\longrightarrow$ ND - The coefficient feeder registers receive bits 0-47 and 59 from the input registers at this time.

10) CLEAR NM - The exponent adder feeder registers are cleared upon receipt of the "Transmit" signal from the Scoreboard.

11) TRANSMIT RESULTS - The 60-bit result of the Add unit is sent to register entry control upon receipt of the "Transmit" signal from the scoreboard.

7.4.5 EXPONENT CIRCUITRY

The exponent circuitry discussed at this point accomplishes the following tasks:

1) Subtracts the exponent of Xk from that of Xj to:
   a) form a 7-bit shift count
   b) select the coefficient with the smaller exponent for gating through the right shift network.
   c) select the larger exponent as the base exponent of the result.
2) Subtract $60_{(8)}$ from the base exponent in the event that the Double Precision mode is selected.
3) Add +1 to the result exponent if a right shift one place is required to correct a coefficient overflow.

The Xj - Xk subtracting network must accomplish more than the simple generation of a difference. It must also generate the absolute value of the difference for use as a shift count (i.e. a negative difference must be made positive). The circuitry which accomplishes this is shown in Figure 7.4-4. To the left, on K31 - K34 are the feeder registers for the Xj - Xk subtractor which contain the exponents in true value form.. (Bit 58 of the original operands is extended to bit 59) Three outputs per stage are used by the subtractor circuitry. For stage 2°, these are pins 21, 26 and 28 which translate as follows:

$$\text{pin } 21 \Longrightarrow \overline{Xk2^\circ}$$

$$\text{pin } 26 \Longrightarrow \overline{Xj2^\circ}$$

$$\text{pin } 28 \Longrightarrow Xj2^\circ + \overline{Xk2^\circ} \text{ (or, } \overline{\overline{Xj} \cdot Xk})$$

Stages 1 and 2 have similar translations for their respective positions.

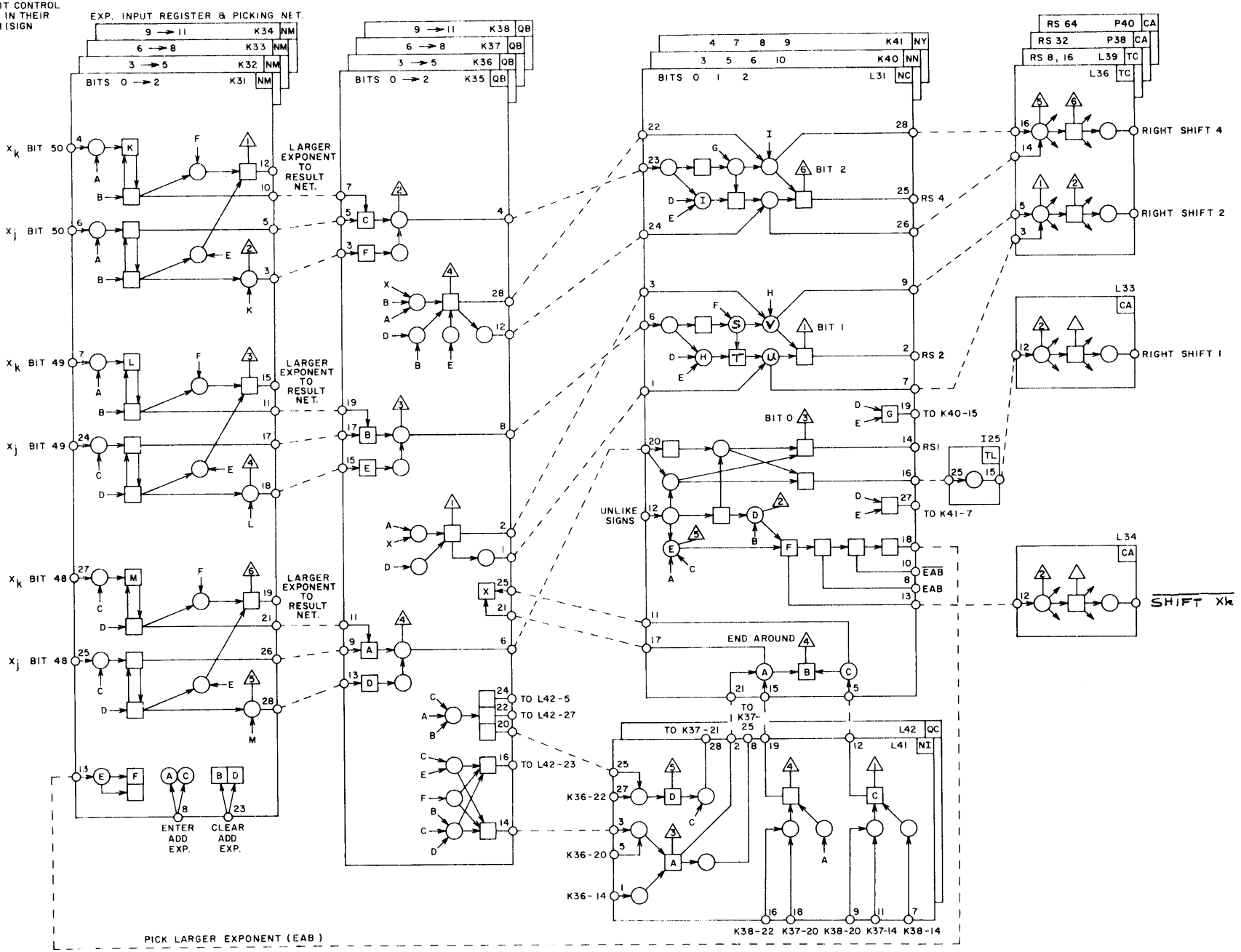

EXPONENT BITS COME DIRECTLY FROM EXIT CONTROL (CH 8) AND ARRIVE IN THEIR TRUE-VALUE FORM (SIGN POSITIVE)
EXP. INPUT REGISTER & PICKING NET.
9 → 11 K34 NM
6 → 8 K33 NM
3 → 5 K32 NM
BITS 0 → 2 K31 NM
9 → 11 K38 QB
6 → 8 K37 QB
3 → 5 K36 QB
BITS 0 → 2 K35 QB
4 7 8 9 K41 NY
3 5 6 10 K40 NN
BITS 0 1 2 L31 NC
RS 64 P40 CA
RS 32 P38 CA
RS 8, 16 L39 TC
L36 TC
X_k BIT 50
X_j BIT 50
X_k BIT 49
X_j BIT 49
X_k BIT 48
X_j BIT 48
LARGER EXPONENT TO RESULT NET.
BIT 2
BIT 1
BIT 0
RS 4
RS 2
RS1
TO K40-15
TO K41-7
I25
RIGHT SHIFT 4
RIGHT SHIFT 2
RIGHT SHIFT 1
L33 CA
L34 CA
UNLIKE SIGNS
END AROUND
EAB
EAB
SHIFT Xk
TO L42-5
TO L42-27
TO L42-23
TO K37-21
TO K37-25
L42 QC
L41 NI
K36-22
K36-20
K36-14
K38-22 K37-20 K38-20 K37-14 K38-14
ENTER ADD EXP.
CLEAR ADD EXP.
PICK LARGER EXPONENT (EAB)

Figure 7.4-4

The above translations are tied to the QB modules (i.e. K35) where equivalence checks and borrow generation take place. Simple Boolean manipulation will prove that the outputs of test points 4, 3 and 2 translate as $\overline{\text{EQUIVALENCE}}$ (exclusive OR) for stages 0, 1 and 2, respectively. These translations are sent to the final summation networks on L31, along with borrow inputs to the individual stages. Before analyzing the borrow generation and propagation logic, a pencil and paper subtraction of 2153 minus 2064 is given:

SUBTRACT: Xj = 0153
Xk = 0064
DIFFERENCE = 0067

(Note that the exponents are unpacked)

IN BINARY:

Xj = 000 001 101 011
Xk = 000 000 110 100
DIFFERENCE = 000 000 110 111

Recall, that when subtracting a "1" from a "0", a borrow is generated. A borrow can be satisfied when subtracting a "0" from a "1". Hence, the following definitions for Borrow, Satisfy and Enable are derived:

BORROW	SATISFY	ENABLE
0 -1	1 -0	0 or 1 -0 -1

The above operands are shown subtracted by the machine (logical) method:

		* * *			
STAGE DEFINITION	=	EEE	EES	EGS	GSS
Xj	=	000	001	101	011
Xk	=	000	000	110	100
STAGE EQUIVALENCE	=	111	110	100	000
BORROW IN	=	000	001	101	000
$(EQ \cdot B) + (\overline{EQ} \cdot \overline{B})$	=	000	000	110	111
IN OCTAL	=	0	0	6	7

(Asterisk indicates those stages with Borrow In)

Borrow generation and group propagation takes place on the QB modules (Figure 7.4-4). To generate a Borrow into stage 2', two possibilities exist: 1) Stage 2° is a generate or 2) Stage 2° is not a satisfy and an End Around Borrow was generated. As always, term "X" of the QB module is the EAB condition. A stage 2° generate is defined by term D, which translates as: $\overline{Xj} \cdot Xk$. No satisfy in stage 2° is the interpretation for term "A", which translates as : $\overline{\overline{Xj} \cdot \overline{Xk}}$. These three conditions are logically combined at test point 1 to yield the following formula:

$$\text{Borrow} \longrightarrow 2' \Longrightarrow (2^\circ = G) + (EAB)(2^\circ \neq S)$$

or, more specifically,

$$B \longrightarrow 2^1 \Longrightarrow (Xj2^\circ \cdot Xk2^\circ) + (EAB)\overline{(Xj2^\circ \cdot \overline{Xk2^\circ})}$$

"Borrow In" translations for the remaining stages can be obtained by applying the above principles. The formulas will become more detailed for the more significant bit positions since more possibilities for borrow generation and propagation exist.

The final borrow and equivalence summations are made on modules L31, K40 and

K41. The logic looks for the conditions,

$$(\text{EQUIVALENCE})\ (\text{BORROW}) + (\overline{\text{EQUIVALENCE}})\ (\overline{\text{BORROW}})$$

to generate a sum of "1". The opposite conditions yield a sum of "0". It is at this point that the sign of the result must be considered, since if it is negative, the complement of the difference will yield the absolute (unsigned) value of the shift count. Four examples follow which show four possible combinations of the Xj and Xk signs and End Around Borrow generation.

1) UNLIKE SIGNS and EAB

Xj = 0010

Xk = 7772
Diff = 0015 = + 15

2) LIKE SIGNS and $\overline{\text{EAB}}$

Xj = 0240

Xk = -0230
Diff = 0010 = + 10

3) UNLIKE SIGNS and $\overline{\text{EAB}}$

Xj = 7772

Xk = -0012
Diff = 7762 = - 15

4) LIKE SIGNS and EAB

Xj = 0230
Xk − 0240
Diff = 7767 = - 10

Note that in cases 1 & 2 the difference is positive and in true form. This indicates that Xj $\geq$ Xk. In cases 3 & 4, the difference is negative and in complement form. This indicates that Xj $<$ Xk. For cases 3 & 4 then, the difference must be complemented to the shift network; in cases 1 & 2 the difference is used as is. To summarize in Boolean,

$$(\overline{LIKE})(EAB) + (LIKE)(\overline{EAB}) \Longrightarrow Xj \geq Xk$$

therefore: 1) shift Xk to the right

2) use the true difference as the shift count

3) use the exponent of Xj as the base exponent.

$$(LIKE)(EAB) + (\overline{LIKE})(\overline{EAB}) \Longrightarrow Xj < Xk$$

therefore: 1) Shift Xj to the right

2) Use the complement of the difference as the shift count

3) Use the exponent of Xk as the base exponent.

The logic for the above conditions is shown on L31 (Figure 7.4-4).

1) TP4 $\Longrightarrow$ EAB

2) terms "A" and "B" together $\Longrightarrow$ $\overline{EAB}$

3) pin 12 $\Longrightarrow$ unlike signs

4) a ZERO out of term "E" $\Longrightarrow$ $(LIKE)(\overline{EAB})$

5) a ZERO out of term "D" $\Longrightarrow$ $(\overline{LIKE})(EAB)$

6) term "F" $\Longrightarrow$ term "E" or term "D"

$(LIKE)(\overline{EAB}) + (\overline{LIKE})(EAB) \Longrightarrow Xj \geq Xk$

L31, pin 13 (same as term "F") goes to L34 which fans out to enable the Xj (if pin 13 = 0) or Xk (if pin 13 = 1) coefficient to the shift network. L34, pin 18 goes back to the feeder registers and select either the Xj (if pin 18 = 0) or Xk (if pin 18 = 1) exponent to the result network.

Selection of the difference, Xj - Xk, must now be made and gated to the shift network. This also is shown on L31. It has been determined that if Xj $\geq$ Xk, the true difference is the shift count; if Xj $<$ Xk, the complement of the difference is the shift count. The complement of the shift count is taken by complementing the EQUIVALENCE term that enters the final summation network if Xj $<$ Xk. Using the subtract logic explained earlier, the following example yields a negative result:

In octal:

Xj	=	0044
Xk	=	0056
Diff	=	7765 = -11

Machine Method:

		***	***	***	***	
Stage definition	=	EEE	EEE	EEG	EGE	
Xj	=	000	000	100	100	
Xk	=	000	000	101	110	
Equivalence	=	111	111	110	101	
Borrow In	=	111	111	111	111	
$(EQ \cdot B) + (\overline{EQ} \cdot \overline{B})$	=	111	111	110	101	
In Octal	=	7	7	6	5	= -12

(Asterisk indicates those stages with Borrow In)

If a sum of "1" is indicated by

$(EQ \cdot \overline{B}) + (\overline{EQ} \cdot B)$ instead of $(EQ \cdot B) + (\overline{EQ} \cdot \overline{B})$,

the answer will be in true value form:

$$\overline{\text{Equivalence}} = 000\ 000\ 001\ 010$$

$$\text{Borrow In} = \underline{111\ 111\ 111\ 111}$$

$$(EQ \cdot B) + (EQ \cdot B) = 000\ 000\ 001\ 010$$

$$\text{Octal} = 0\ \ 0\ \ 1\ \ 2$$

As a logic example, L31, TP1 will be analyzed. First, recall that if $Xj \geq Xk$, term "F" = 1. Term F is ANDed with the condition, $\overline{\text{EQUIVALENCE}}$, at inverter "S". Terms "D" & "E" in combination indicate that $Xj < Xk$. These conditions are ANDed with the condition, EQUIVALENCE, at inverter "H". Term "T" ORs terms "S" and "H" to yield the translation:

$$(Xj \geq Xk)(\overline{EQ}) \quad + \quad (Xj < Xk)(EQ)$$

Term "T" is then ANDed with pin 1 ($\overline{\text{BORROW}}$) at term "U". A "zero" out of term "U" yields the following translation:

$$\left[\overline{\text{BORROW}}\right] \quad \left[(Xj \geq Xk)(\overline{EQ}) \quad + \quad (Xj < Xk)(EQ)\right]$$

which summarizes the two of the four possible ways to generating a sum of one.

Inverter "V" summarizes the result for the BORROW condition. This translation is left to the reader since in principle the same as inverter "U".

In summary, the true value of the sum will always be seen at test points 3, 1 and 6 to indicate a RS1, RS2,or RS4 condition. These conditions are fanned out on TC and CA modules.

Figure 7.4-5 shows the exponent adder and result network of the Add Unit. The purposes of this logic are to 1) subtract $60_{(8)}$ from the exponent during double precision mode, 2) gate the larger exponent to the transmitters for single precision mode and 3) to add 1 to either the single or double precision exponent if overflow occurs in the coefficient requiring it to be right shifted.

All the above operations are accomplished by performing a difference operation; therefore, the circuit acts as a subtractor. If Double precision is specified, the circuit subtracts $60_{(8)}$ or $57_{(8)}$ (depending on coefficient overflow) from the larger exponent. In the single precision mode with coefficient overflow, "one" is added to the base exponent by subtracting the complement, thereby adding. If overflow does not occur, the base exponent is gated as the result. In summary, one of the following events occurs:

Single Precision & $\overline{\text{Overflow}}$ $\Longrightarrow$ Gate base exponent unaltered.

Single Precision & Overflow $\Longrightarrow$ Subtract 7776 (Add 1)

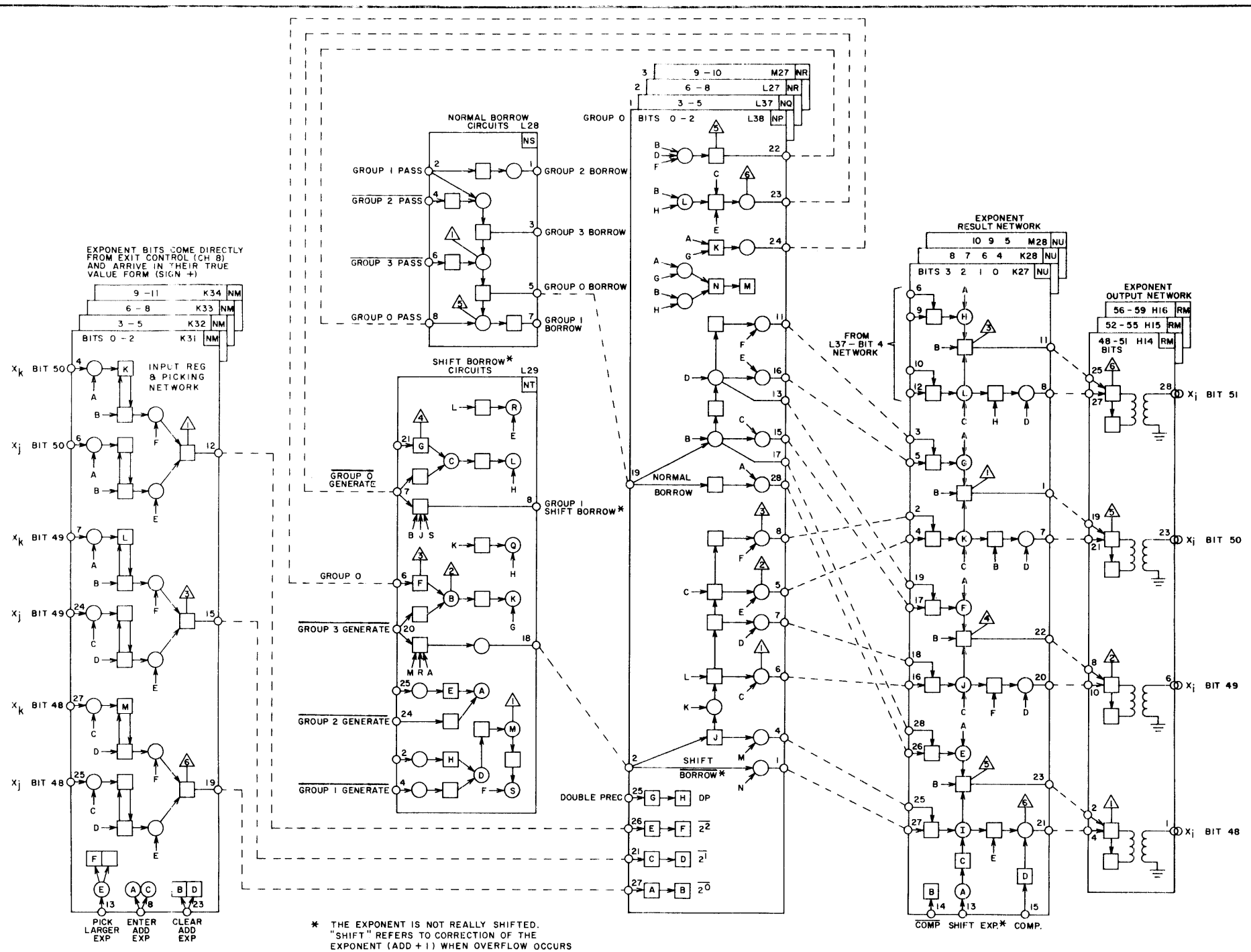
Double Precision & $\overline{\text{Overflow}}$ $\Longrightarrow$ Subtract 0060 (Subtract 60)

Double Precision & Overflow $\Longrightarrow$ Subtract 0057 (Subtract 57)

The following examples illustrate the four cases (base exponent in all cases = $0060_{(8)}$):

1) SP & $\overline{\text{O'flow}}$

Base Exponent
and Final Exponent = 000 000 110 000



* THE EXPONENT IS NOT REALLY SHIFTED. "SHIFT" REFERS TO CORRECTION OF THE EXPONENT (ADD + 1) WHEN OVERFLOW OCCURS IN THE COEFFICIENT.

Figure 7.4-5

2) SP & O'flow:

Stage Definition	GGG GGG EEG GGE
Base Exponent	000 000 110 000
Subtract 7776	111 111 111 110
FINAL EXPONENT	000 000 110 001

3) DP & $\overline{\text{O'flow}}$:

Stage Definition	EEE EEE EEE EEE
Base Exponent	000 000 110 000
Subtract 0060	000 000 110 000
FINAL EXPONENT	000 000 000 000

4) DP & O'flow:

Stage Definition	EEE EEE ESG GGG
Base Exponent	000 000 110 000
Subtract 0057	000 000 101 111
FINAL EXPONENT	000 000 000 001

In any of the above cases (except the first), a Generate occurs if the base exponent contains a "zero" and the corresponding bit of the value being subtracted is a "one". In the second case, a generate may occur in any stage except 2^0, since 2^0 of the subtrahend bits are "0". In case four, a generate may occur only in stages 2^0. 2^1, 2^2, 2^3 or 2^5. These facts should be kept in mind when analyzing the borrow generation and propagation logic. The exponent result network and transmitters are shown to the right of Figure 7.4-5. Note that the two inputs to the transmitters will enable either the true or complemented result exponent, as selected on the NU modules by terms "B" and "D". Also, the overflow and no overflow selection is made on these modules. The bit "0" result, for example, may be gated through terms "E", in the no overflow ($\overline{\text{Shift}}$) case or through term "I" in the overflow case.

A "1" in bit 2° of the result during overflow cases occurs if either pin 25 or 27 is a "0". Translating pins 4 and 1 of L38 for "0" yields the following formulas:

$$L38,4 = \text{"0"} \Longrightarrow B(SP \cdot \overline{2^\circ} + DP \cdot 2^\circ)$$

$$L38,1 = \text{"0"} \Longrightarrow \overline{B}(SP \cdot 2^\circ + DP \cdot \overline{2^\circ})$$

Associating these formulas with examples 2 & 4 above, will yield the proper result in bit 2°:

Single Precision:

		$\overline{\text{Borrow in}}$	Borrow in
2°	=	1	0
minus	=	0	0
RESULT	=	1	1

Double Precision:

	Borrow in	$\overline{\text{Borrow in}}$
	1	0
	1	1
	1	1

The remaining four cases for SP and DP overflow cases yield a result = 0:

Single Precision:

		Borrow in	$\overline{\text{Borrow in}}$
2°	=	1	0
minus	=	0	0
RESULT	=	0	0

Double Precision:

$\overline{\text{Borrow in}}$	Borrow in
1	0
1	1
0	0

As indicated earlier, the no overflow results for 2° are entered via term "E" on K27. Pins 28 or 26 must equal "0" to yield a result of "1". Hence, pins 17 or 28 of L38 must equal "0". These pins translate as follows:

$$\text{pin 17} = \text{"0"} \Longrightarrow \overline{2^\circ} \quad (DP \cdot B)$$

$$\text{pin 28} = \text{"0"} \Longrightarrow 2^\circ \quad (SP + \overline{B})$$

Relating these formulas to examples 1 and 3 above yield the proper results:

Single Precision

Borrows are disabled and bit 2^0 is gated unaltered via pin 28 (pin 17 always = "1")

Double Precision

Borrow in =	1	0	0	1
2^0	1	0	1	0
minus	0	0	0	0
RESULT =	0	0	1	1

The remaining bit positions are handled in a similar manner, but in some of the higher positions, generation of borrows must be gated by the single or double precision modes. For example, the single and double precision overflow cases differ in the second octal digit. In single precision, bit 2^4 will generate if 2^4 of the exponent = "0", since a "1" is being subtracted (Example #2). For double precision, bit 2^4 can never generate since a "0" is always subtracted from bit 2^4 (Example #4). Analysis of the remaining bit positions can be accomplished by keeping the above concepts in mind.

7.4.6. RIGHT SHIFT NETWORK

A seven rank right shift network is used to shift the coefficient with the smaller exponent to the right. The seven ranks shift 1, 2, 4, 8, 16, 32 or 64 places. A shift count is given by the true value of the difference of the Xj and Xk exponents. (See section 7.4.5). As the operand passes through each rank, it will be right shifted or unshifted, depending upon whether or not the corresponding bit of the shift count is set.

Figure 7.4-6 is a representative logic diagram of the shift network. The first rank of the network as well as the coefficient selection circuitry is shown on K21. Selection of the Xj or Xk coefficient takes place after determining which of the exponents was smaller. (Section 7.4.5). In the first rank the selected coefficient may be unshifted (term B=1) or right shifted one place (term E=1) depending on whether or not bit 20 of the shift count is set. The remaining six ranks are located, two ranks per module, on M32, M42 and P33.

Each rank has two possible inputs per stage from the previous rank (i.e. a bit is either shifted or not shifted). Both possibilities are fed to a given stage as an ORed input and again may be shifted, or unshifted in that rank, depending on the state of the corresponding bit of the shift count. Since it is not possible for a given rank to shift and at the same time not shift, only one of the two inputs to a stage can be present at a time.

As a logic example, assume that Xj is to be shifted $32_{(10)}$ places. In

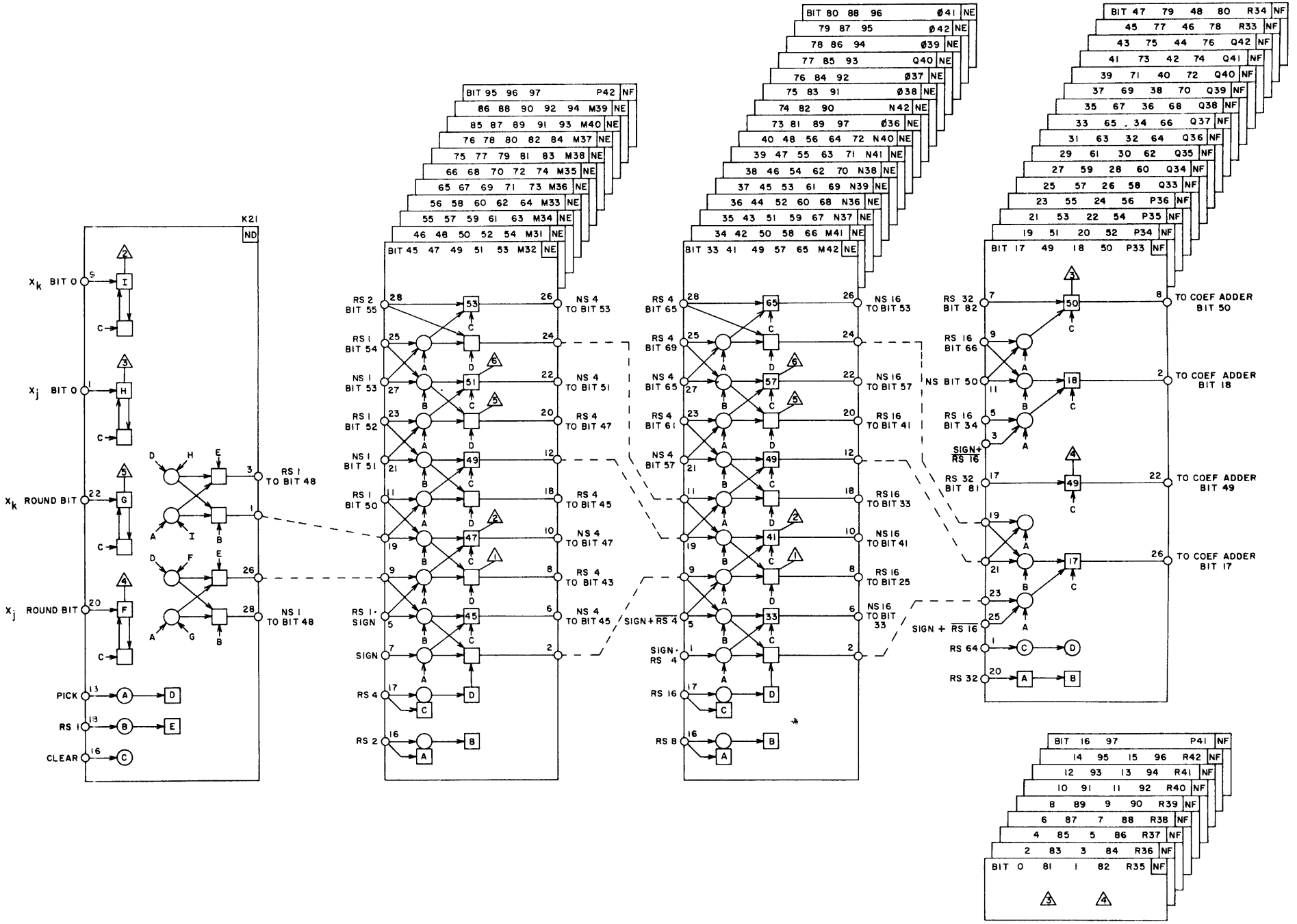

K21
ND
x_k BIT 0
x_j BIT 0
x_k ROUND BIT
x_j ROUND BIT
PICK
RS 1
CLEAR
RS 1 TO BIT 48
NS 1 TO BIT 48
BIT 95 96 97 P42 NF
86 88 90 92 94 M39 NE
85 87 89 91 93 M40 NE
76 78 80 82 84 M37 NE
75 77 79 81 83 M38 NE
66 68 70 72 74 M35 NE
65 67 69 71 73 M36 NE
56 58 60 62 64 M33 NE
55 57 59 61 63 M34 NE
46 48 50 52 54 M31 NE
BIT 45 47 49 51 53 M32 NE
RS 2 BIT 55
RS 1 BIT 54
NS 1 BIT 53
RS 1 BIT 52
NS 1 BIT 51
RS 1 BIT 50
RS 1 SIGN
SIGN
RS 4
RS 2
NS 4 TO BIT 53
NS 4 TO BIT 51
RS 4 TO BIT 47
RS 4 TO BIT 45
NS 4 TO BIT 47
RS 4 TO BIT 43
NS 4 TO BIT 45
BIT 80 88 96 Ø41 NE
79 87 95 Ø42 NE
78 86 94 Ø39 NE
77 85 93 Q40 NE
76 84 92 Ø37 NE
75 83 91 Ø38 NE
74 82 90 N42 NE
73 81 89 97 Ø36 NE
40 48 56 64 72 N40 NE
39 47 55 63 71 N41 NE
38 46 54 62 70 N38 NE
37 45 53 61 69 N39 NE
36 44 52 60 68 N36 NE
35 43 51 59 67 N37 NE
34 42 50 58 66 M41 NE
BIT 33 41 49 57 65 M42 NE
RS 4 BIT 65
RS 4 BIT 69
NS 4 BIT 65
RS 4 BIT 61
NS 4 BIT 57
RS 4 BIT 57
SIGN + RS 4
SIGN · RS 4
RS 16
RS 8
NS 16 TO BIT 53
NS 16 TO BIT 57
RS 16 TO BIT 41
RS 16 TO BIT 33
NS 16 TO BIT 41
RS 16 TO BIT 25
NS 16 TO BIT 33
BIT 47 79 48 80 R34 NF
45 77 46 78 R33 NF
43 75 44 76 Q42 NF
41 73 42 74 Q41 NF
39 71 40 72 Q40 NF
37 69 38 70 Q39 NF
35 67 36 68 Q38 NF
33 65 34 66 Q37 NF
31 63 32 64 Q36 NF
29 61 30 62 Q35 NF
27 59 28 60 Q34 NF
25 57 26 58 Q33 NF
23 55 24 56 P36 NF
21 53 22 54 P35 NF
19 51 20 52 P34 NF
BIT 17 49 18 50 P33 NF
RS 32 BIT 82
RS 16 BIT 66
NS BIT 50
RS 16 BIT 34
SIGN+ RS 16
RS 32 BIT 81
SIGN + RS 16
RS 64
RS 32
TO COEF ADDER BIT 50
TO COEF ADDER BIT 18
TO COEF ADDER BIT 49
TO COEF ADDER BIT 17
BIT 16 97 P41 NF
14 95 15 96 R42 NF
12 93 13 94 R41 NF
10 91 11 92 R40 NF
8 89 9 90 R39 NF
6 87 7 88 R38 NF
4 85 5 86 R37 NF
2 83 3 84 R36 NF
BIT 0 81 1 82 R35 NF

Figure 7.4-6

this case, bit 2° of Xj (actually, 2^{49} with respect to the result) should be shifted to bit 2^{17}. The shift count will be $40_{(8)}$ ($100\ 000_{(2)}$) to enable only shift rank 32. For the remaining ranks, the No Shift (NS) gate will be high. K21, TP3 is the feeder register for Xj 2°. If the Xj $<$ Xk gate is high, term "D" is a "one" to enable Xj 2° into rank 1. Assuming that 2° is a "one" and because the $\overline{RS1}$ gate (term B) is a one, pins 1 and 3 of K21 are both ones. Pin 1 feeds pin 19 of M32 which is ANDed with pin 11 and term A ($\overline{R2}$) to yield a "zero" out of the circle and a one out of pin 12. This makes M42, pin 19 a "one". ANDing with pin 11 and term "A" ($\overline{R8}$) forces a "zero" out of the circle and a one out of pin 12. This makes P33 pin 21 a "one" which, when ANDed with pin 19 and term B (R32) yields a "one" out of Test Point 5. Pin 26 feeds bit 2^{17} of the adder.

Note that in rank 64, the bits shown are not sent to the adder for Right Shifts of 64 places. This is because right shifting any of these bit positions would cause that bit to be lost. The RS64 does enable shifting bit positions higher than 2^{63}.

7.4.7 COEFFICIENT ADDER

The coefficient adder logic is shown in Figure 7.4-7. Also included is the circuitry which right shifts the coefficient one place in the event that overflow occurs. Generates, Satisfies and Enables are defined as follows:

Generate	Satisty	Enables
0	1	1 0
0	1	0 1

The final summation logic yields a sum of "1" for the following conditions:

$$(\text{EQUIVALENCE})(\overline{\text{CARRY}})+(\overline{\text{EQUIVALENCE}})(\text{CARRY})$$

The following example illustrates the adder logic operation:

```
ADD:   Xj  =  224547
       Xk  =  143715
      SUM  =  370464
```

```
                          *** **       * *    *
STAGE DEFINITION          GEE EEG EEE SES EGE SES
                Xj    = 010 010 100 101 100 111
                Xk    = 001 100 011 111 001 101
      EQUIVALENCE     = 100 001 000 101 010 101
            CARRY       111 110 000 001 100 001
(EQ'·C) + (EQ·C')       011 110 000 100 110 100
```

The two operands are entered into the NK (or NG) modules shown at the left in Figure 7.4-7. Bit 2^5 of the unshifted coefficient is entered at pin 18 of Q20. The input from the shift network is pin 12 and 8. Terms B and F reflect the true value of the operands

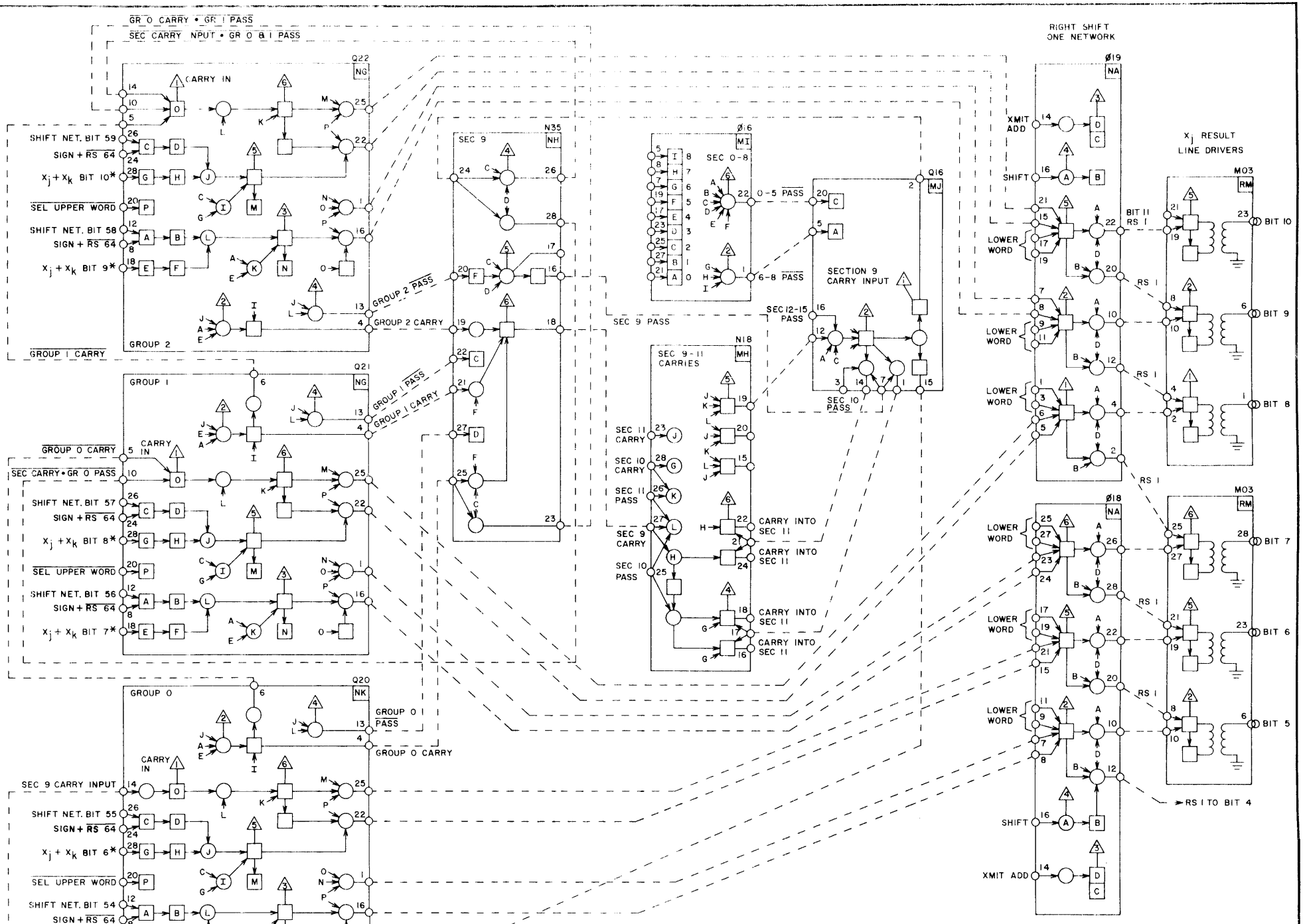

GR 0 CARRY • GR 1 PASS
SEC CARRY INPUT • GR 0 & 1 PASS
RIGHT SHIFT ONE NETWORK
Q22
NG
CARRY IN
SHIFT NET. BIT 59
SIGN + RS 64
$X_j + X_k$ BIT 10*
SEL UPPER WORD
SHIFT NET. BIT 58
SIGN + RS 64
$X_j + X_k$ BIT 9*
GROUP 2
GROUP 2 PASS
GROUP 2 CARRY
GROUP 1 CARRY
N35
NH
SEC 9
SEC 9 PASS
Ø16
MI
SEC 0-8
0-5 PASS
6-8 PASS
Q16
MJ
SECTION 9 CARRY INPUT
SEC12-15 PASS
SEC 10 PASS
N18
MH
SEC 9-11 CARRIES
SEC 11 CARRY
SEC 10 CARRY
SEC 11 PASS
SEC 9 CARRY
SEC 10 PASS
CARRY INTO SEC 11
CARRY INTO SEC 11
CARRY INTO SEC 11
CARRY INTO SEC 11
Q21
NG
GROUP 1
GROUP 1 PASS
GROUP 1 CARRY
GROUP 0 CARRY
SEC CARRY • GR 0 PASS
CARRY IN
SHIFT NET. BIT 57
SIGN + RS 64
$X_j + X_k$ BIT 8*
SEL UPPER WORD
SHIFT NET. BIT 56
SIGN + RS 64
$X_j + X_k$ BIT 7*
Q20
NK
GROUP 0
GROUP 0 PASS
GROUP 0 CARRY
CARRY IN
SEC 9 CARRY INPUT
SHIFT NET. BIT 55
SIGN + RS 64
$X_j + X_k$ BIT 6*
SEL UPPER WORD
SHIFT NET. BIT 54
SIGN + RS 64
$X_j + X_k$ BIT 5*
Ø19
NA
XMIT ADD
SHIFT
LOWER WORD
X_j RESULT LINE DRIVERS
M03
RM
BIT 11 RS 1
BIT 10
BIT 9
BIT 8
Ø18
NA
BIT 7
BIT 6
BIT 5
RS 1
RS 1 TO BIT 4
* THESE BITS ARE OF THE COEFFICIENT WITH THE LARGER EXPONENT.

Figure 7.4-7

and a "0" out of term L indicates that both bits are set. A "0" out of term K indicates that both bits are cleared. A "1" out of test point 3 therefore indicates equivalence in this stage of the adder.

Generation and propagation of carries is not dealt with in detail since at this point in the course of study, the student should have a clear idea of this circuitry. It is sufficient to indicate that the carry is entered from the propagation networks at pin 14 of Q20, making test point 1 (term "Ø") a "1".

The carry and equivalence conditions are combined with the circuits feeding pins 1 and 16. If term "Ø" = 1 (Carry) and "N" = 1 ($\overline{\text{EQUIVALENCE}}$) and "P" = 1 (Select Upper Word) pin 1 will be a zero. Also, if "Ø" = 0, TP3 = 1 and "P" = 1 pin 16 will be a zero. Either of these cases cause TP2 on Ø18 to be a "1" causing a sum of "1" to be transmitted on bit 5 (or bit 4 if right shifted) of the data trunk. The original formula for a sum of "1" is therefore proven. i.e.

$$(\text{EQUIVALENCE})(\overline{\text{CARRY}}) + (\overline{\text{EQUIVALENCE}})(\text{CARRY})$$

of course, if the above conditions are not met, a sum of "0" is transmitted.

The right shift one network (i.e.Ø 18) is enabled if coefficient overflow occurred. This makes term B = 1 and when the "Transmit" signal is received causes the result to be right shifted. (i.e. bit 5 to bit 4, etc.) The absence of the Overflow condition

causes the unshifted result to be transmitted.

If double precision was selected, a "Select Lower Word" signal would be generated to cause selection of the lower 48-bits of the sum. This result is entered into the right shift one network (i-e. pins 9 & 11) to be transmitted unshifted (no overflow) or shifted right one place (overflow).

7.4.8 OVERFLOW/INDEFINITE/INFINITE

Non-standard operand forms may be generated as a result of testing the original Xj and Xk exponents or by generation of overflow or underflow during the addition (subtraction) process.

In testing original operand, the logic looks for infinite and indefinite values of Xj and Xk as shown in Figure 7.4-8. The Xk test circuitry is in the upper left corner. It tests Xk for a 1777 or 3777 configuration in bits 48 through 58 (exponent bits 0 - 10) and stores the condition in flip-flops on F20 and F21. Xj is tested in the same manner with the circuitry shown in the lower left corner. The conditions of Xj are also stored in flip-flops on F20 and F21. If an operand is indefinite or infinite, the condition is transmitted to chassis #5 via H06.

If either Xj or Xk are infinite (3777) this condition is stored in F26, TP2. An overflow condition is forced and a shift count of 112 ($160_{(8)}$)is generated. The shift count will cause the shifted coefficient to be completely lost since the right shift is end-off. The lower 48 bits of the 96-bit sum will therefore be all zeros.

An infinite result can be generated in three ways:

1) Xj = 1777

2) Xk = 1777

3) Xj = 3777 and Xk = 3777 and signs are unlike

These cases are logically combined on module Ø32, TP4, which translates as:

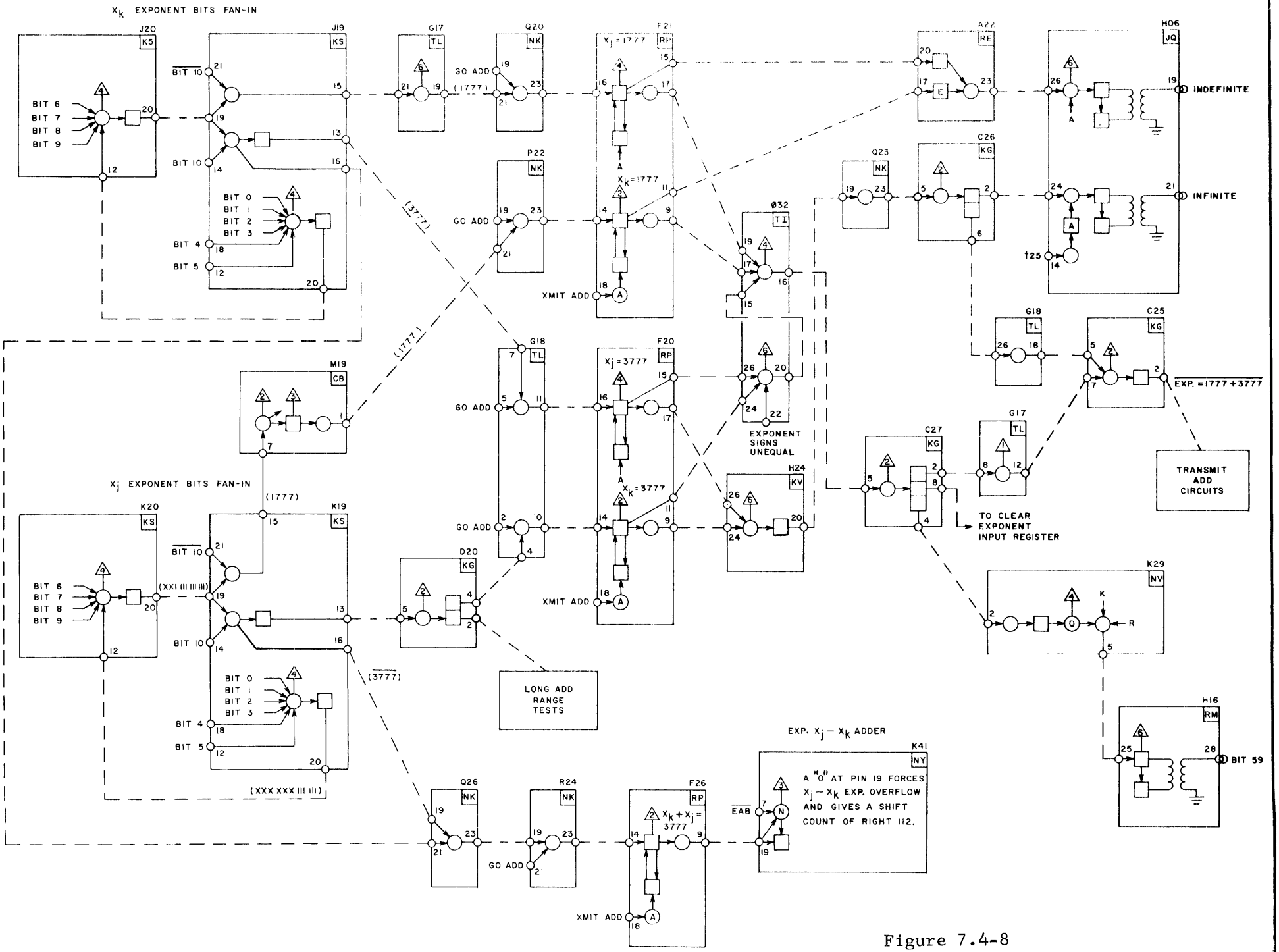

X_k EXPONENT BITS FAN-IN
X_j EXPONENT BITS FAN-IN
J20
K5
J19
KS
G17
TL
Q20
NK
F21
RP
X_j = 1777
A22
RE
H06
JQ
INDEFINITE
INFINITE
BIT 6
BIT 7
BIT 8
BIT 9
BIT 10
BIT 0
BIT 1
BIT 2
BIT 3
BIT 4
BIT 5
GO ADD
(1777)
P22
NK
X_k = 1777
XMIT ADD
Q23
C26
KG
t25
Ø32
TI
G18
F20
X_j = 3777
X_k = 3777
C25
EXP. = 1777 + 3777
M19
CB
EXPONENT SIGNS UNEQUAL
H24
KV
C27
G17
TRANSMIT ADD CIRCUITS
TO CLEAR EXPONENT INPUT REGISTER
K20
K19
(XXI III III III)
(XXX XXX III III)
(3777)
D20
KG
LONG ADD RANGE TESTS
K29
NV
H16
RM
BIT 59
EXP. $X_j - X_k$ ADDER
K41
NY
A "0" AT PIN 19 FORCES $X_j - X_k$ EXP. OVERFLOW AND GIVES A SHIFT COUNT OF RIGHT 112.
EAB
Q26
R24
F26
$X_k + X_j$ = 3777

Figure 7.4-8

(Xj = 1777) + (Xk = 1777) + (Xj = 3777)(Xk = 3777)(SIGNS UNEQUAL)

This condition is fanned out on module C27 and clears the exponent input registers and disables setting bit 59 of the result register. Module C25 ORs the infinite and indefinite conditions; if either is present, transmitting the coefficient bits is disabled.

Exponent overflow resulting from addition of operands can result only if the base exponent (larger exponent) was 3776 and coefficient overflow occurs. In this case, 1 is added to 3776 to compensate for a right shift of the coefficient. If this condition occurs, transmission of the coefficient occurs along with the final exponent of 3777.

Underflow may occur if the larger exponent has a large negative magnitude and Double Precision is selected. In this case, $60_{(8)}$ must be subtracted from the base exponent. The following is an example of the underflow case:

Add in Double Precision:

Xj = 0043 X---------X

Xk = 0023 X---------X

1) Xj exponent is the larger (more positive)

2) Subtract $60_{(8)}$ (D.P.) from base exponent.

$$\begin{array}{rl} 0043 & = -3734 \\ \text{D.P.} \Longrightarrow & \underline{- \quad 60_{(8)}} \\ & -4014 \end{array}$$

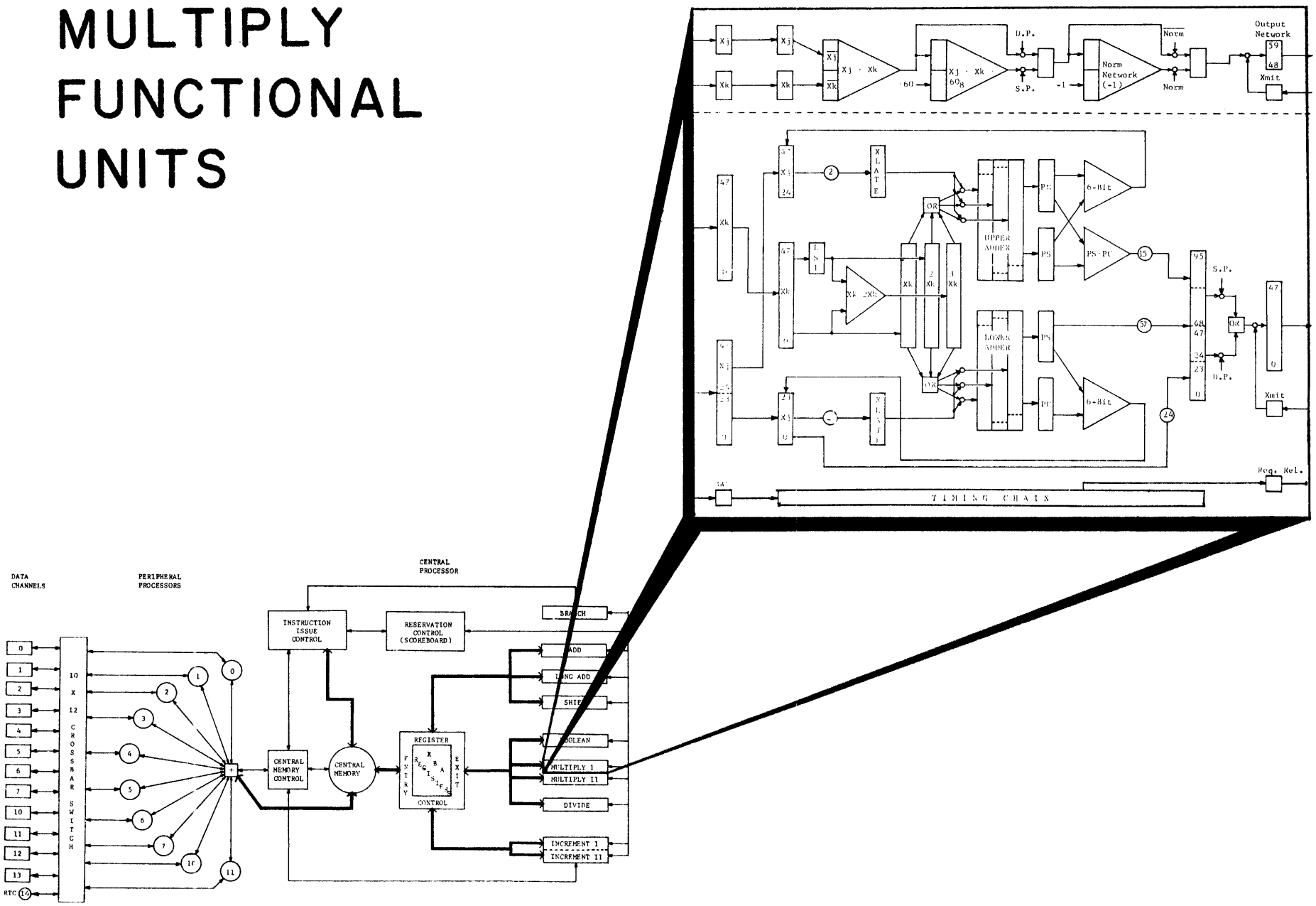
3) The result is more negative than -3777 and therefore, an underflow case. The circuitry which makes the underflow test is shown on sheet 224 of the Customer Engineering Diagrams. A "0" out of N27, test point 2 indicates that the "Transmit" has been received and that Underflow has not occurred. It allows the final exponent to be sent on the data trunk. If overflow had occurred, neither the complement or $\overline{\text{complement}}$ gates (K29) would be high and all zeros would be sent in the upper 12 bit positions, indicating that underflow had occurred.

SECTION 7.5

MULTIPLY

FUNCTIONAL UNITS

MULTIPLY FUNCTIONAL UNITS



THE MULTIPLY FUNCTIONAL UNITS

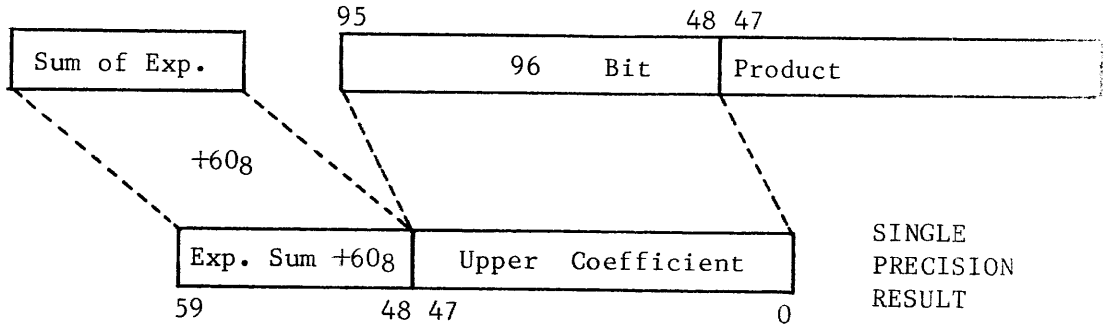
7.5.1 INTRODUCTION

GENERAL: The 6600 Central Processor contains two Multiply functional units designated Multiply I and Multiply II. Duplication of the units occurs because multiplication is one of the slowest, yet most frequently used processes in mathematical or scientific applications. Since the two units operate in parallel, single and double precision results may be obtained almost simultaneously. (Refer to Appendix A for an explanation of 6000 Series single/double precision.)

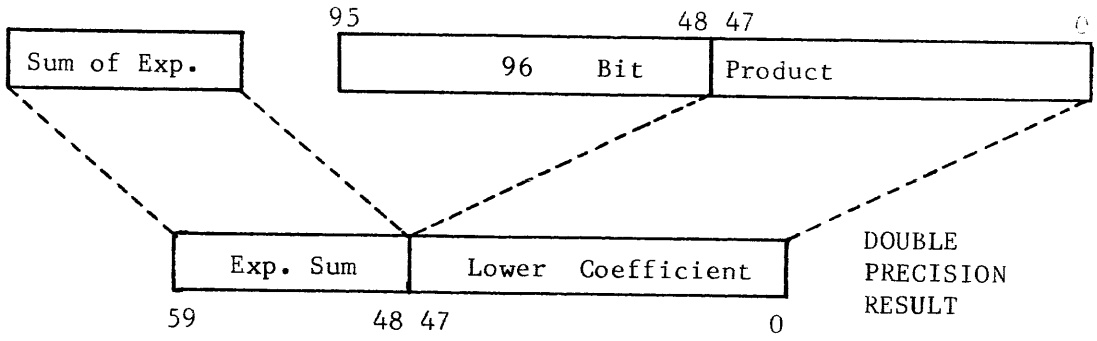
The following instructions select a Multiply unit and permit unrounded single or double precision and rounded single precision calculations to be made:

40	FXi = Xj * Xk	Single Precision Product
41	RXi = Xj * Xk	Rounded Single Precision Product
42	DXi = Xj * Xk	Double Precision Product

The Multiply units generate 96-bit products from two 48-bit coefficients. If single precision is selected, the upper 48-bits of the product and the sum of the exponents plus 60_8 are returned as the result. The addition of 60_8 is necessary since, in selecting the upper half of the 96-bit result, the binary point is effectively moved from the right of bit 2^0 to the right of bit 2^{48}. In other words, the product magnitude was decreased by 2^{48} and a corresponding increase of the exponent is required.



If double precision is selected, the lower 48-bits of the product and the sum of the exponents are returned as the result:



A normalized product will result only if both source operands were normalized. Two cases must be considered in discussing the normalize process: (See Figure 7.5-1)

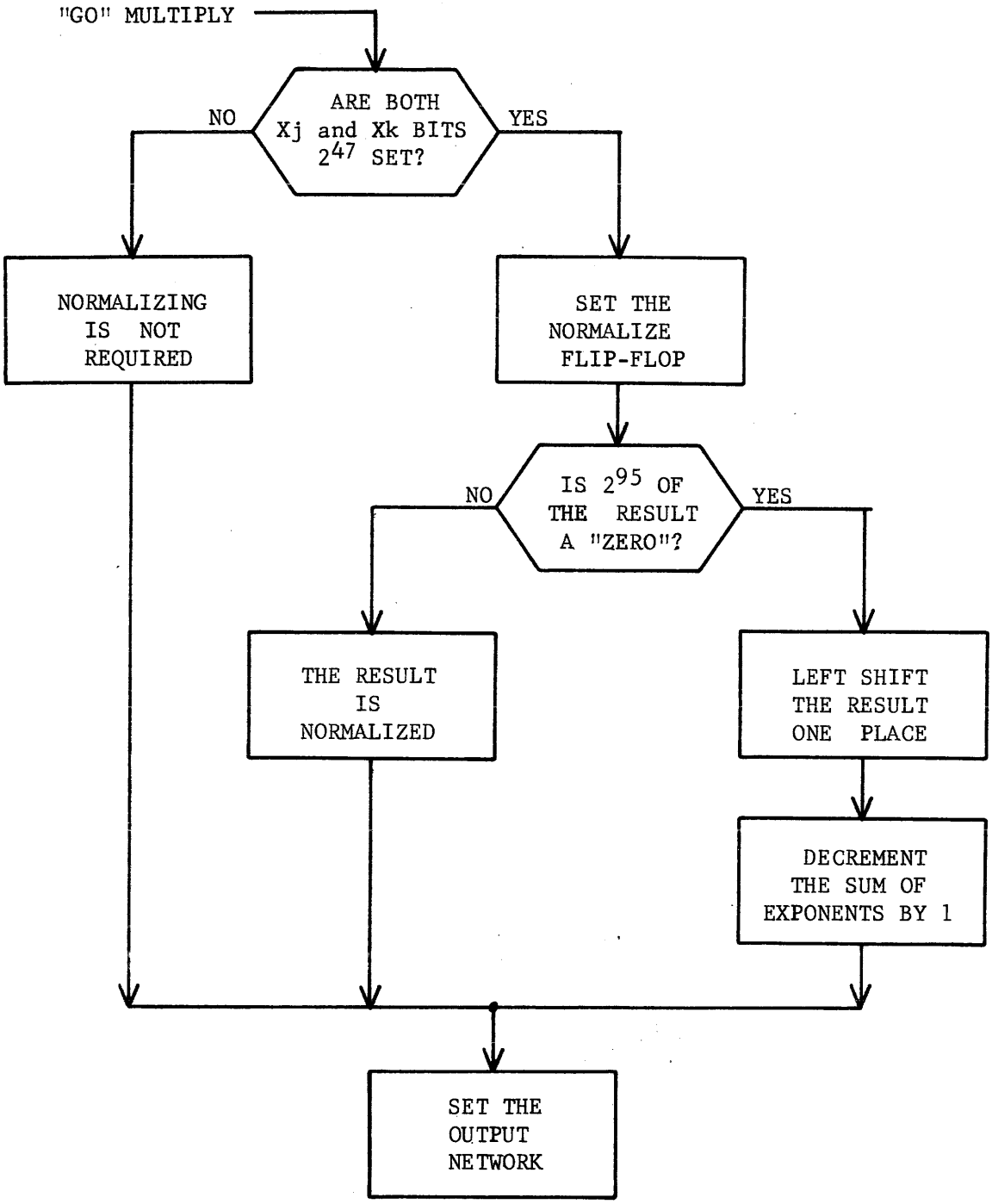
CASE 1 - Left Shift One is Required

$$\begin{array}{r} 4000000000000000 \\ 4000000000000000 \\ \hline 20000000000000000000000000000000 \end{array}$$

For this case, the multiply result must be left shifted one place to yield the following normalized result:

40000000000000000000000000000000

Since the coefficient is increased 2^1 times by this manipulation, the exponent must be decremented by one. Thus, if a



MULTIPLY NORMALIZE FLOW CHART

Figure 7.5-1

single precision multiply occurs and a left shift is required to normalize the coefficient, the final exponent is:

Sum of Exponents + 60_8 - 1, or

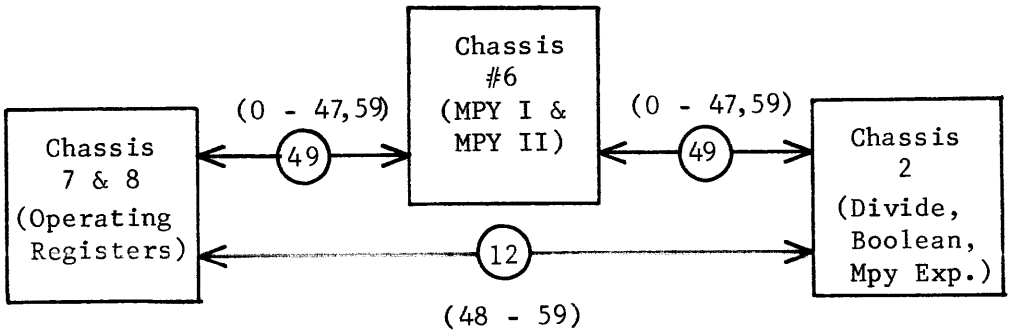
Sum of Exponents + 57_8

CASE 2 - Result is Already Normalized

$$\begin{array}{r} 7000000000000000 \\ 7000000000000000 \\ \hline 6100000000000000000000000000000000 \end{array}$$

In this case, the most significant bit position of the result (2^{95}) is a "one" as a result of the multiply process, and a left shift is not required. Hence the final exponent for single precision is the sum of the exponents + 60_8.

The Multiply functional units share data trunk #2 with the Divide and Boolean units. Multiply 1 and 2 hold second and third priorities (respectively) for reading operands; third and fourth (respectively) for storing results. Since all of chassis 6 is utilized to contain the coefficient logic of both Multiply units, the exponent logic is on chassis 2 with the Divide and Boolean units. Hence, data paths on this trunk look as follows:



The lower 48-bits (coefficient) and sign bit are sent to and manipulated on chassis #6, while the upper 12-bits (exponent and signs) are sent directly to chassis #2 where the final exponent is generated. (The lower 48-bits are also sent to chassis #2 via chassis #6, but are used only on chassis #6 during multiply operations.)

THE MULTIPLY PROCESS (COEFFICIENT)

The following example illustrates the familiar, every-day method of multiplication:

EXAMPLE #1:

Multiplicand	1234
Multiplier	1234
First Partial Product	5160
Second Partial Product	3724
Third Partial Product	2470
Fourth Partial Product	1234
FINAL PRODUCT	1547420

It can be seen that the final product is the result of adding the four partial products.

A slightly different method is employed in the following example. The multiplier is split in half and two multiplications are performed yielding two partial sums. The final product is obtained by adding the two partial sums. This final process will here-after be referred to as "MERGE."

EXAMPLE #2:

FIRST MULTIPLICATION	SECOND MULTIPLICATION
1234	1234
1200	0034
247000	5160
1234	3724
1503000 = 1st Partial Sum	44420 = 2nd P. Sum

MERGE:

```
   44420
1503000
-------
1547420  =  FINAL PRODUCT
```

The 6600 Multiply units use this method in their operation. The multiplier is, of course, 48-bits long instead of 12-bits as in the previous examples. Hence, when split in half, two 24-bit segments result. Since two 24-bit multiplications (referred to as "Upper" and "Lower") take place in parallel, the multiply time is $\frac{1}{2}$ that required for a single 48-bit multiply. Because the Merge operation cannot take place until after two partial sums are generated, the overall multiply time is somewhat greater than $\frac{1}{2}$, but never-the-less, substantially reduced.

The hardware performs the split multiplies by examining two bits of the multiplier at a time. The following steps show the sequence of multiplication:

a) Examine the lower two bits of each multiplier and thereby determine whether 0, 1, 2 or 3 times the multiplicand must be added.

i.e.
```
00 = 0X   (X = times)
01 = 1X
10 = 2X
11 = 3X
```

b) Add the indicated multiple of the multiplicand to the previous partial product.

c) Left shift the new partial product two bit positions.

d) Examine the next two bit positions of the multiplier and perform steps b and c again.

e) Repeat step d until all bits have been examined.

f) Merge the two partial products generated by steps a - e and form the final product.

The original multiply example will be used to illustrate the above procedure. Each repitition is referred to as a "PICK" (since this is the terminology used in the 6600 hardware documents).

EXAMPLE #3:

PROBLEM:
```
   1234
   1234
1547420
```

1) Form 0, 1, 2 and 3 times the multiplicand.

```
0X = 0000(8) = 000 000 000 000(2)
1X = 1234    = 001 010 011 100
2X = 2470    = 010 100 111 000   (formed by left shifting the
                                  multiplier one bit position)
3X = 3724    = 011 111 010 100   (formed by adding 1X and 2X)
```

2) Use two 18-bit registers to hold the split multiplier and the partial products:

UPPER PP	UPPER MULT.
000 000 000 000	001 010

LOWER PP	LOWER MULT.
000 000 000 000	011 100

3) Use two 12-bit adders to add the partial products and the selected multiple of the multiplicand.

4) Pick #1:

a) Translate the lower two bits of the upper and lower registers.

UPPER = 10 = 2X LOWER = 00 = 0X

b) Add partial products (initially all zeros) to the selected multiple of the multiplicand.

```
        UPPER                    LOWER
PP = 000 000 000 000     PP = 000 000 000 000
2X = 010 100 111 000     0X = 000 000 000 000
     ---------------          ---------------
     010 100 111 000          000 000 000 000
```

c) Put the results of each adder back into the respective partial sum register and right shift the register two places.

UPPER		LOWER	
000 101 001 110	000 010	000 000 000 000	000 111

5) Pick #2:

a) Translate the lower two bits

UPPER = 10 = 2X LOWER = 11 = 3X

b) Add the partial sums to the selected multiple of the multiplicand:

UPPER		LOWER	
PS =	000 101 001 110	PS =	000 000 000 000
2X =	010 100 111 000	3X =	011 111 010 100
	011 010 000 110		011 111 010 100

c) Place results in partial sum registers and right shift two places:

UPPER		LOWER	
000 110 100 001	100 000	000 111 110 101	000 001

6) Pick #3:

a) Translate the lower two bits:

UPPER = 00 = 0X LOWER = 01 = 1X

b) Add partial sums to the selected multiple of the multiplicand:

PS =	000 110 100 001	PS =	000 111 110 101
0X =	000 000 000 000	1X =	001 010 011 100
	000 110 100 001		010 010 010 001

c) Place results in partial sum registers and right shift two places:

UPPER		LOWER	
000 001 101 000	011 000	000 100 100 100	010 000
0 1 5 0	$3\ 0_{(8)}$	0 4 4 4	$2\ 0_{(8)}$

7) Merge:

At this point, the two final partial products have been generated and must be merged to form the final product. Since the upper partial product is 6-bits more significant than the lower, it is off-set 6 places to the left and then added:

LOWER PARTIAL PRODUCT	=			000	100	100	100	010 000
UPPER PARTIAL PRODUCT	=	000	001	101	000	011	000	
FINAL PRODUCT	=	000	001	101	100	111	100	010 000
OCTAL EQUIVALENT	=	0	1	5	4	7	4	2 0

The final product agrees with the products generated in the earlier examples.

The previous example is the basic method of multiplication used in the 6600. It can be seen that execution time of the multiply in the example is determined largely by the time spent performing the additions for each partial sum. The 6600, using a 48-bit multiplier, would require 12 iterations of the above type to complete the multiply (checking four bits of the multiplier with each iteration). The adder time alone would prohibit the desired execution time of one micro-second. The problem is resolved by the use of a special three level adder which is not a full adder, but a "Carry Save Adder."

The concept of this adder eliminates the time consuming carry and satisfy checks which are typical of full adders. Instead, each level has two outputs, Pseudo Sums and Pseudo Carries. To illustrate the concept of the adder, they are defined as follows: *

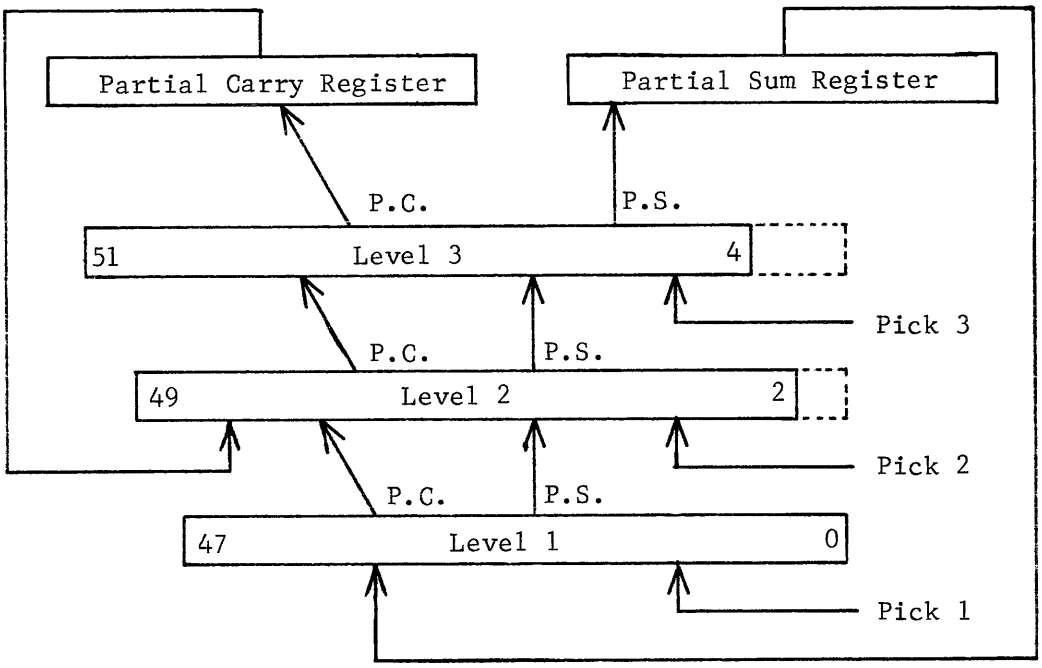
Pseudo Sums	1	0	1	0
	0	1	1	0
	1	1	0	0
Pseudo Carries	1	0	1	0
	0	1	1	0
	0	0	1	0

(Pseudo carries will be displaced one bit to the left since they affect the next significant bit position)

Each level has several inputs: 1) Pseudo sums from the previous level, 2) Pseudo carries from the previous level, and 3) 0, 1, 2 or 3 times the multiplicand, as specified by the configuration of 2 bits of the multiplier (defined as a "Pick"). The following diagram illustrates the 3-level, carry save adder.

---

* Actually, there is somewhat more to the definition of the inputs and outputs of the adder. At this point, only the Carry-save concept is illustrated. For the detailed analysis of the 3-level adder, refer to Section 7.5.6.



THREE LEVEL ADDER

It can be seen that with one iteration (three picks) of the adder, six bits of the multiplier are manipulated. Since two such adders are used in parallel (Upper and Lower), 12-bits are handled with each iteration. Hence, four iterations are required to multiply 48-bit operands and one additional iteration is used to perform the Merge operation.

Note that the Pick 2 and Pick 3 inputs are displaced 2 bits to the left with reference to the previous level since each pick is 2 bits more significant than the last. Also, the Partial Sums and Carries generated by the third level are displaced 6 bit positions to the left when fed back to the first level during the second, third and

fourth iterations. In other words, bit 2^6 of the P.C. and P.S. registers feeds bit 2^0 of the first level, 2^7 feeds 2^1, etc. This occurs since each iteration (3 picks) is 6 bits more significant than the previous one.

The following example illustrates the 3-level adder concept. The operands are the same as those used in the previous examples. Since the multiplier is only 12-bits long, only one iteration and the merge operation will be required. In the example, the terms "Partial Sum" and Partial Carry" refer to the final contents of the P.S. and P.C. registers; "Pseudo Sum" and "Pseudo Carry" refer to the sum and carry outputs of each level.

PROBLEM:

$$\begin{array}{r} 1234 \\ \underline{1234} \\ 1547420 \end{array}$$

LOWER ITERATION:

$$\begin{array}{r} 1234 \\ \underline{34} \\ 44420 \end{array}$$

$$\begin{array}{lcll} 0X & = & 0000_{(8)} & = 000\ 000\ 000\ 000_{(2)} \\ 1X & = & 1234 & = 001\ 010\ 011\ 100 \\ 2X & = & 2470 & = 010\ 100\ 111\ 000 \\ 3X & = & 3724 & = 011\ 111\ 010\ 100 \end{array}$$

$$\begin{array}{lcll} \text{Pick \#1} & = & 0X & = 000\ 000\ 000\ 000 \\ \text{Pick \#2} & = & 3X & = 011\ 111\ 010\ 100 \\ \text{Pick \#3} & = & 1X & = 001\ 010\ 011\ 100 \end{array}$$

```
                    000 000 000 000   Partial Sum
                    000 000 000 000   Partial Carry
                    000 000 000 000   Pick 1  (0X)
                    000 000 000 000   Pseudo Sum
                    000 000 000 000   Pseudo Carry
                 01 111 101 010 0     Pick 2  (3X)
                 01 111 101 010 000   Pseudo Sum
                 00 000 000 000 000   Pseudo Carry
              0 010 100 111 00        Pick 3  (1X)
              0 011 011 010 010 000   Partial Sum
              0 001 001 010 000 000   Partial Carry
                            010 000   FULL ADD

                              2   0   Octal
```

Note that by fully adding the final partial sums and carries the partial product, 44420, can be generated. Recall, that the Merge operation must then take place and this requires a full add of the Upper and Lower Partial Products. In order to save more time, the partial sums and carries are saved and will be used during Merge to generate the final product. But at this point, the addition of the lower six bits of P.S. and P.C. will yield the lower two octals of the final product, 20. This is accomplished by a special 6-bit adder for each iteration of the multiply step.

```
UPPER ITERATION:        1234
                          12
                      150 3000
```

```
Pick #1  =  2X  =  010 100 111 000
Pick #2  =  2X  =  010 100 111 000
Pick #3  =  0X  =  000 000 000 000
```

	000	000	000	000	Partial Sum
	000	000	000	000	Partial Carry
	010	100	111	000	Pick 1 (2X)
	010	100	111	000	Pseudo Sum
	000	000	000	000	Pseudo Carry
01	010	011	100	0	Pick 2 (2X)
01	000	111	011	000	Pseudo Sum
00	100	001	000	000	Pseudo Carry
0 000	000	000	00		Pick 3 (0X)
0 001	100	110	011	000	Partial Sum
0 000	000	010	000	000	Partial Carry

The final partial sums and carries of the Upper Iteration will also be saved and used during the Merge operation.

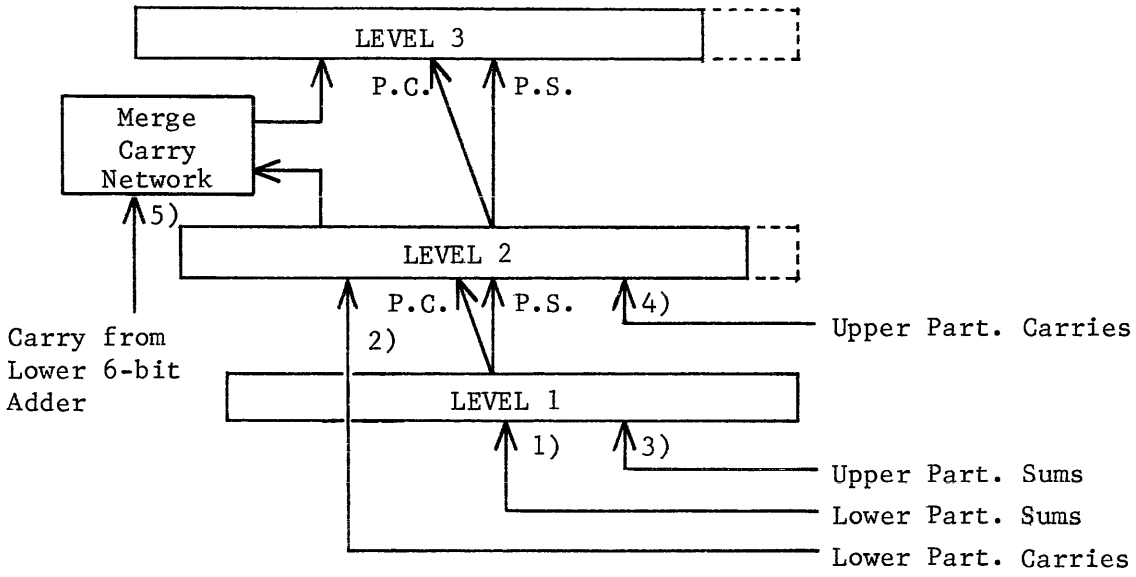
MERGE:

Because the results of the Upper and Lower iterations are in the form of Partial Carries and Partial Sums, the Merge logic must manipulate the following five values:

1) Lower partial sums.
2) Lower partial carries.
3) Upper partial sums.
4) Upper partial carries.
5) A possible carry into bit 2^6 of the result from the lower six-bit adder.

The lower three level adder will be used during the Merge phase of Multiply to handle the five inputs. Since each level has only three inputs, the five values are fed into the levels as follows:

LOWER THREE LEVEL ADDER DURING MERGE



Recall, that during normal iterations, the output of the third level was in the form of partial sums and partial carries. The Merge operation requires the result to be fully added. This is accomplished by the "Merge Carry Network" which essentially converts level three into a full adder. It summarizes the carries and satisfies for each stage and feeds the carries into the proper stages of the third level. It is not used during the normal iterations because of the extra time involved in checking satisfies, carries, etc.

The following illustrates the Merge portion of the example. Recall, that the lower 6-bits of the final product have been generated by fully adding the lower 6-bits of the Lower partial sum and partial carry registers. The result was $20_{(8)}$.

	000	011	011	010	Lower Partial Sum
	000	001	001	010	Lower Partial Carry
01	100	110	011	000	Upper Partial Sum
01	100	100	001	000	Pseudo Sum
00	000	110	110	100	Pseudo Carry
00	000	010	000	000	Upper Partial Carry
01	100	000	111	100	Pseudo Sum
00	001	100	000	000	Pseudo Carry
				0	Carry, Lower 6-bit Add
01	101	100	111	100	FULL ADD
1	5	4	7	4	Octal

When the lower six bits are appended, the final product results:

1547420 = FINAL PRODUCT

The intent of the previous example was to show the concept of the three-level adder and the Merge operation. The actual multiplication of two 48-bit operands requires four iterations of the type explained and then the merge operation. Each iteration generates 6-bits of the final result, so after four passes through the adders, the lower 24-bits of the final product will have been generated. The upper adder generates 18-bits of a partial product (partial because it may be modified by a carry from the 24-bit lower product) with three iterations, but on the fourth iteration saves all partial sums and carries for use during the merge phase.

As was mentioned, the lower 3-level adder is used during the merge operation, but with the use of 48-bit operands, its inputs will be:

1) 18-bits of the upper partial product (from the first three iterations.

2) upper partial sums (from the fourth iteration).

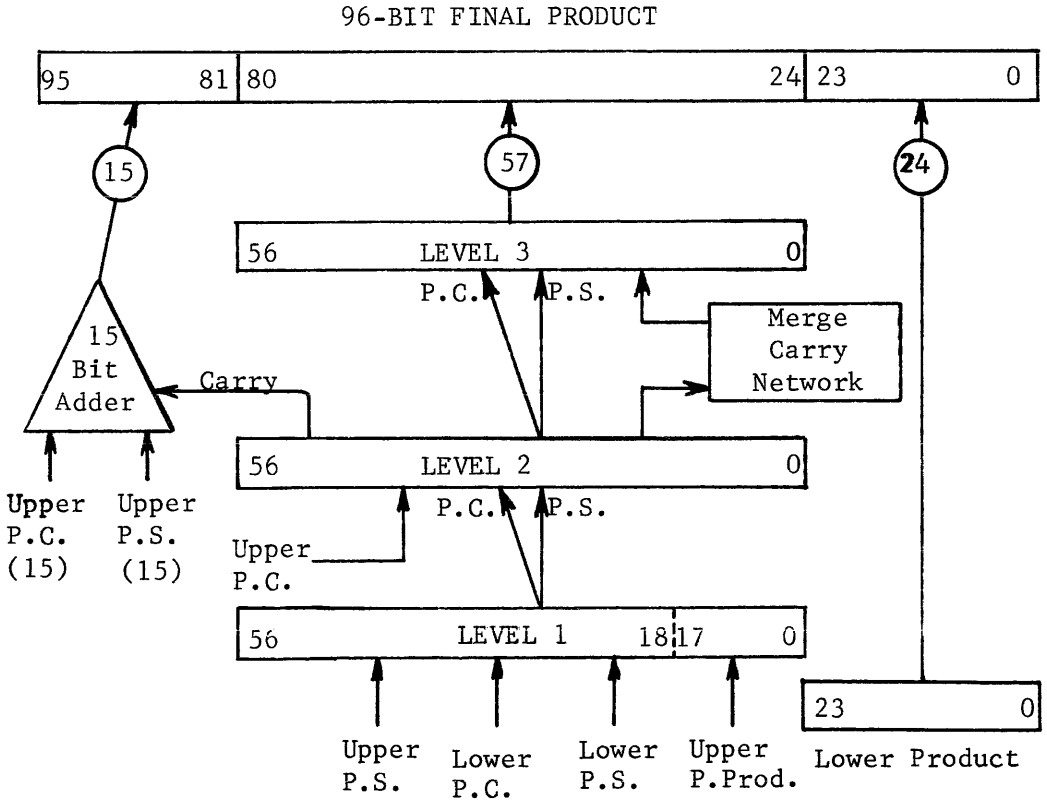
3) upper partial carries (from the fourth iteration).

4) lower partial sums (from the fourth iteration).

5) lower partial carries (from the fourth iteration).

6) carry from the lower 6-bit adder (after the fourth 6-bit addition).

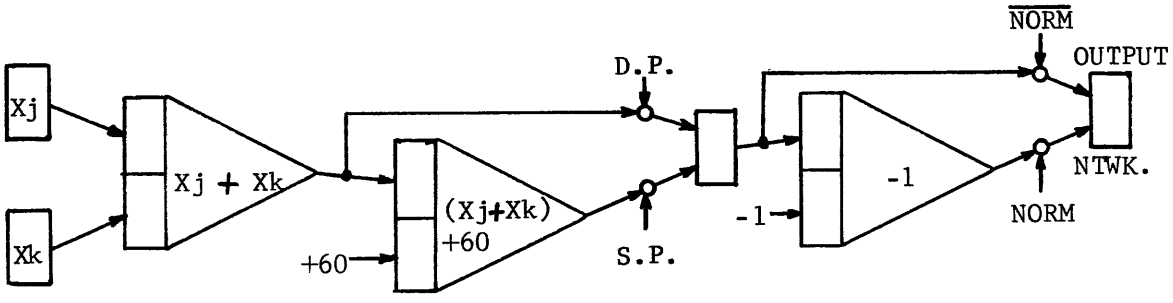
During merge, the lower 3-level adder will generate 57-bits of the final product. The upper 15-bits of the final product are produced by a special 15-bit adder (used only during merge) which adds the upper 15-bits of the upper partial sums and carries and a possible carry from the three-level adder. In general, the merge looks as follows:



EXPONENT FORMATION:

Since the exponent logic for the multiply units is located on chassis 2, communication between chassis 6 and 2 is required. For example, chassis 6 must select a single or double precision exponent, or an exponent reflecting the normalization of the coefficient. Chassis 2 must order an error coefficient if it detects an error condition during exponent formation.

Three adders are required for the formation of the final exponent. The first adder forms the algebraic sum of the exponents of Xj and Xk. This result will be used if double precision is selected. The second adder adds $60_{(8)}$ to the sum of the exponents - its result is used if single precision is selected. The third adder subtracts one from the result of the first (if D.P.) or the second (if S.P.) adder in the event that a left shift one place is required to normalize the coefficient. The following diagram represents the exponent adders in block form:



MULTIPLY EXPONENT LOGIC - CHASSIS 2

In forming the sum of the Xj and Xk exponents, the following sequence occurs. In order that all cases of negative and positive exponents are illustrated, the following numerical examples will be used:

Xj	=	+05	-05	-05	+05
Xk	=	+13	-13	+13	-13
RESULT	=	+20	-20	+06	-06

STEP 1 Obtain the true value of the exponents. This is accomplished by complementing the upper 12-bits if bit 59 is a "1" (negative coefficient). If bit 59 is a "0", the exponent is already in true form. Hence, the true values of the exponents in packed form are:

2005	1772	1772	2005
2013	1764	2013	1764

STEP 2 Unpack the exponents by:

a) Extending bit 58 into bit 59 of the feeders:

6005	1772	1772	6005
6013	1764	6013	1764

b) Complementing the exponent magnitude bits (48 - 57) into the feeders. The feeders now hold the complemented, unbiased exponents:

7772	0005	0005	7772
7764	0013	7764	0013

STEP 3 Add. The results are:

7757	0020	7771	0006

STEP 4 Complement the result. (Since the operands are in complement form in the feeders):

0020 7757 0006 7771

STEP 5 Set or clear bits 59 and 58 according to the following rules (as determined by the exponent error check logic):

SET BIT 59 if:

a) Neither Xj nor Xk were zero operands (i.e. 48 - 59 = $0000_{(8)}$).

b) AND neither Xj nor Xk were indefinite operands (i.e. 1777).

c) AND the final sign of the coefficient is negative. This is determined by the following rules:

1) like signs $\Longrightarrow$ positive result

2) unlike signs $\Longrightarrow$ negative result

d) AND underflow was not generated

e) AND a "Transmit" was received from the Scoreboard.

CLEAR BIT 59 if the above conditions are not met.

BIT 58 = TRUE adder output (adders 1, 2 or 3) if:

a) Neither Xj nor Xk were indefinite operands

b) AND one or both of the operands were infinite (3777) OR overflow was generated.

BIT 58 = COMPLEMENT of adder output (1, 2 or 3) if:

a) Xj or Xk or Both were indefinite operands

b) OR neither Xj nor Xk were infinite operands AND overflow was not generated.

Assuming that no error conditions exist, and the final sign of the coefficient is positive, bit 59 will be cleared and the complement of the adder result bit 58 will be taken. Thus, the final configuration of the upper 12-bits will be:

2020 1757 2006 1771

The second (+60) and third (-1) adders are not discussed in detail here since their operations are not difficult to understand. The (Xj + Xk) + 60 Adder simply places the unbiased result of the Xj - Xk Adder in one feeder and a +60 in the other. Hence the sum of the exponents + 60 is formed.

The third adder places the result of the first (for double precision) or the second (for single precision) adder in one feeder and a minus 1 in the other. The result is either the

1) Sum of the exponents minus 1 (DP), or

2) Sum of the exponents plus 57 (SP).

If a left shift one place is required to normalize the result, the output of the third adder is used; if not, the output of adders one or two is used as the final exponent.

The exponent logic is discussed in greater detail in Sections 7.5.10 through 7.5.13. At this point, only the overall picture and "pencil and paper" methods of manipulation need be understood.

7.5.2 INSTRUCTION LIST/DATA FLOW

The following instructions will select one of the Multiply functional units. The terms in parenthesis are the ASCENT symbolic codes used in assembler coding. Data flow can be followed on figure 7.5-1.

40 FLOATING PRODUCT OF Xj and Xk to Xi (FXi = Xj * Xk)

DEFINITION: This instruction multiplies two floating point quantities obtained from operand registers Xj (Multiplier) and Xk (Multiplicand) and packs the upper 48-bits of the 96-bit product in operand register Xi.

The result is unnormalized when either or both operands are unnormalized. In this case the final exponent is the sum of the exponents plus $60_{(8)}$. The result is normalized only when both operands are normalized. The exponent in this case is the sum of the exponents plus $60_{(8)}$ or $57_{(8)}$ (if a left shift one place is required to normalize).

DATA FLOW: Bits 0 - 47 and bit 59 of Xj and Xk are sent to chassis 6 and bits 48 - 59 to chassis 2. On chassis 2, three exponents are formed: 1) Xj + Xk - used for a double precision result not left shifted to normalize, 2) Xj + Xk + 60 used for a single precision result not left shifted to nor-

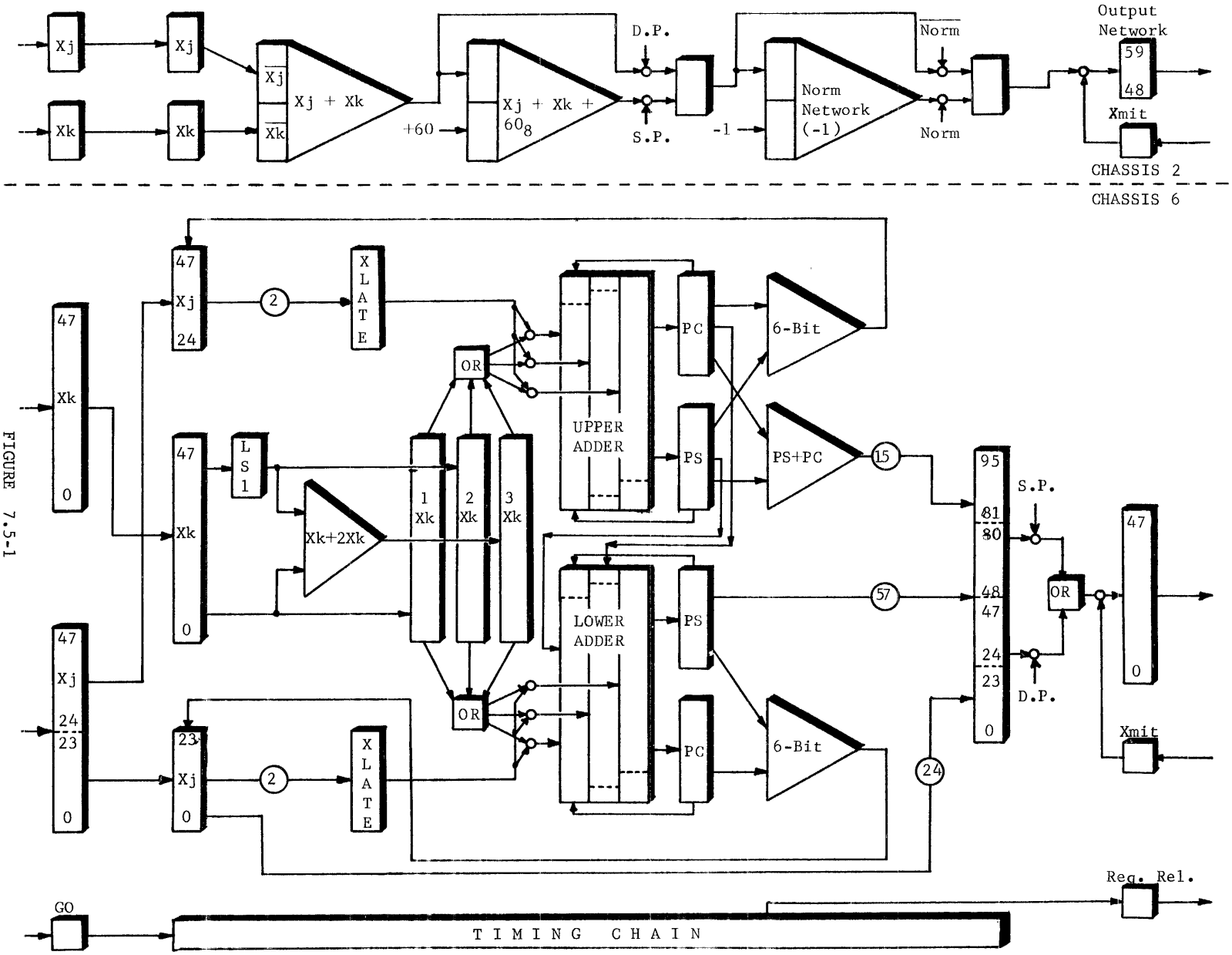

Xj
Xk
Xj + Xk
+60
Xj + Xk + 60₈
D.P.
S.P.
-1
Norm Network (-1)
Norm
Output Network
59
48
Xmit
CHASSIS 2
CHASSIS 6
47
24
0
23
OR
XLATE
LS 1
Xk+2Xk
1 Xk
2 Xk
3 Xk
UPPER ADDER
LOWER ADDER
PC
PS
6-Bit
PS+PC
15
57
24
95
81
80
48
47
23
Reg. Rel.
GO
TIMING CHAIN

Figure 7.5-1.1

malize and 3) Xj + Xk - 1 (D.P.) or Xj + Xk + 57 (S.P.) - used if a left shift one is required to normalize. The final exponent is sent from chassis 2 to Register Entry Control upon receipt of the "Transmit" pulse from the Scoreboard.

Bits 0 - 47 of Xj and Xk are multiplied on chassis 6 (method is explained in Section 7.5.1) where a 96-bit product is formed. The upper 48-bits are selected for Single Precision and are sent to Register Entry Control via chassis 2 upon receipt of the "Transmit" signal.

41 ROUND FLOATING PRODUCT of Xj and Xk to Xi (RXi = Xj * Xk)

DEFINITION: This instruction multiplies two floating point quantities obtained from operand registers Xj (multiplier) and Xk (multiplicand) and packs the upper 48-bits of the 96-bit result in operand register Xi. During the first iteration of the multiply process, a "one" is added to bit 46 of Xk. This, in effect, adds $\frac{1}{4}$ (20 ------ $0_{(8)}$) to the final product.

When both Xj and Xk are normalized, the result will also be normalized. If a left shift one place is required to normalize the final product, a $\frac{1}{2}$ round will occur (since the 20 ------ 0 was doubled by the left shift). If no left shift is required to normalize, or if either of the operands is unnormalized, a $\frac{1}{4}$ round will occur (since the 20 ------ 0 was not left shifted).

As mentioned, the result is unnormalized when either or both of the operands are unnormalized. In this case, the final exponent is the sum of the exponents plus $60_{(8)}$. The result is normalized only when both operands are normalized. In this case the exponent is the sum of the exponents plus $60_{(8)}$ or $57_{(8)}$ (if a left shift was required to normalize).

DATA FLOW: Bits 0 - 47 and bit 59 of Xj and Xk are sent to chassis 6 and bits 48 - 59 to chassis 2. The following three exponents are formed on chassis 2 (See data flow for the 40 instruction):

1) Xj + Xk

2) Xj + Xk + 60

3) Xj + Xk - 1 or Xj + Xk + 57

The final exponent is sent from chassis 2 to Register Entry Control upon receipt of the "Transmit" signal.

Bits 0 - 47 are multiplied on chassis 6 (method explained in Section 7.5.1) where a rounded, 96-bit product is formed. The upper 48-bits are selected and sent to Register Entry Control via chassis 2 upon receipt of the "Transmit" signal.

42 FLOATING DOUBLE PRECISION PRODUCT of Xj and Xk to Xi

(DXi = Xj * Xk)

DEFINITION: This instruction multiplies two floating point

quantities obtained from operand registers Xj and Xk and packs the lower 48-bits of the 96-bit product in operand register Xi. This result is not necessarily a normalized quantity.

The exponent of this result is $60_{(8)}$ less than the exponent resulting from a 40 instruction using the same operands.

DATA FLOW: Bits 0 - 47 and bit 59 of Xj and Xk are sent to chassis 6 and bits 48 - 59 to chassis 2. On chassis 2, the following three exponents are formed (See Data Flow for the 40 instruction):

1) Xj + Xk

2) Xj + Xk + 60

3) Xj + Xk - 1 or Xj + Xk + 57

The final exponent is sent from chassis 2 to Register Entry Control upon receipt of the "Transmit" signal.

Bits 0 - 47 of Xj and Xk are multiplied on chassis 6 (method is explained in Section 7.5.1) where a 96-bit product is formed. The lower 48-bits are selected and sent to Register Entry Control via chassis 2 upon receipt of the "Transmit" signal.

7.5.3 MODE BITS

Two mode bits are necessary to enable the Multiply units to distinquish between the 40, 41 and 42 instructions. They are:

1) Round
2) Double Precision

As with the other functional units, mode bits are transmitted to the unit at Scoreboard issue time.

ROUND:

When a Single Precision Round opcode is processed, rounding-up is desired if the magnitude of the lower 48-bits of the 96-bit product is greater than or equal to ½. If the discarded bits are less than ½, no rounding should occur and the unaltered upper 48-bits will be taken as the final product coefficient. The following examples illustrate the two cases:

1) ROUND-UP NOT REQUIRED:

```
                 5000000000000000
               x 4000000000000001
                 5000000000000000
 240000000000000000000000000000000
 24000000000000005000000000000000
```

Left Shift to Normalize:

```
5000000000000001 2000000000000000
 Final            Less than ½,
 Coefficient      thus, no round
```

2) RUUND-UP IS REQUIRED:

```
                6000000000000000
              x 4000000000000001
                ----------------
                6000000000000000
3000000000000000000000000000000
--------------------------------
30000000000000006000000000000000
```

Left Shift to Normalize:

```
60000000000000014000000000000000
                \______________/
                  ½ or more,
                therefore, round
```

The correct rounded result:

```
6000000000000002xxxxxxxxxxxxxxxx
\______________/\______________/
     Final       Not Available
  Coefficient    with 41 inst.
```

One of the more common methods of rounding is referred to as a post-round. The procedure is to add ½ to the final product, thereby causing a carry into the least significant bit of the product if the discarded bits have a magnitude equal to ½ or greater. The values obtained above are used for illustration:

1) ROUND-UP DOES NOT OCCUR:

```
Final Product   =  50000000000000012000000000000000
Add ½ to Round                     4000000000000000
                   --------------------------------
No Carry Occurs    50000000000000016000000000000000
```

2) ROUND-UP DOES OCCUR:

```
Final Product   =  60000000000000014000000000000000
Add ½ to Round                     4000000000000000
                   --------------------------------
A Carry Occurs     60000000000000020000000000000000
```

The great disadvantage of the post-round method, especially if time is a major consideration in the computer design, is that an extra adder cycle is required to add the rounding factor, ½. The overall multiply time is therefore increased considerably.

The 6600 utilizes a faster method referred to as pre-round. With this method, the rounding factor is added during the first iteration of the multiply process, thereby eliminating the need for an add cycle at the end of the sequence. Since it is not known, at this early point of the sequence, whether or not a left shift of the final product will be required (to normalize), a factor of ¼ (20 ------ 0) is added instead of ½. This prevents the erroneous addition of 1 to the final product, which would be the end effect if ½ was added and the final product was left shifted one place. Thus, if a left shift one place is required to normalize the final product, a ½ round will occur since the rounding factor, 20 ------ 0, is doubled by the left shift. If no left shift is required to normalize, or if either of the operands is unnormalized, a ¼ round will occur since the 20 ------ 0 is not left shifted for these cases. Post and Pre-round are illustrated with the following examples:

1) POST-ROUND:

Xk	=	6000000000000000
Xj	=	x 4000000000000001
1st Partial Product	=	6000000000000000
2nd Partial Product	=	3000000000000000000000000000000
Initial Product	=	3000000000000006000000000000000
L.S. to Normalize	=	6000000000000014000000000000000
½ Rounding Factor	=	4000000000000000
Final Product	=	6000000000000002000000000000000

2) PRE-ROUND:

Xk	=	6000000000000000
Xj	=	4000000000000001
¼ Rounding Factor	=	2000000000000000
1st Partial Product	=	6000000000000000
2nd Partial Product	=	3000000000000000000000000000000
Initial Product	=	3000000000000001000000000000000
L.S. to Normalize	=	6000000000000002000000000000000
yields Final Product		

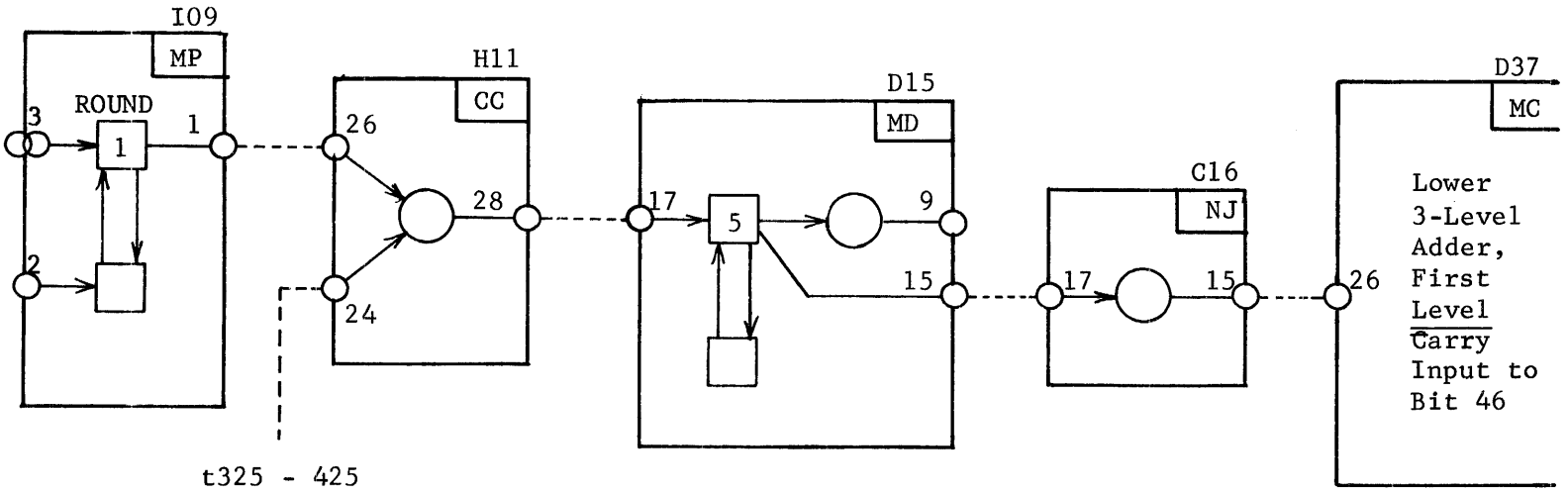
Note that the correct rounded result was obtained with both methods.

A "ROUND" mode bit is transmitted to chassis 6 if fm = 41 is translated from the U2 Register. This causes bit 52 of the Lower Partial Carry Register to be set on module D15 (Figure 7.5-2) during the first iteration of the multiply step (t325 - 425 of the multiply timing chain). Bit 52 of the P.C. Register feeds bit 46 of the first level of the Lower 3-level Adder, and this effectively adds the ¼ rounding factor on the first iteration. Since the P.C. Registers are cleared before each iteration, the round bit will remain for only the first iteration.

DOUBLE PRECISION:

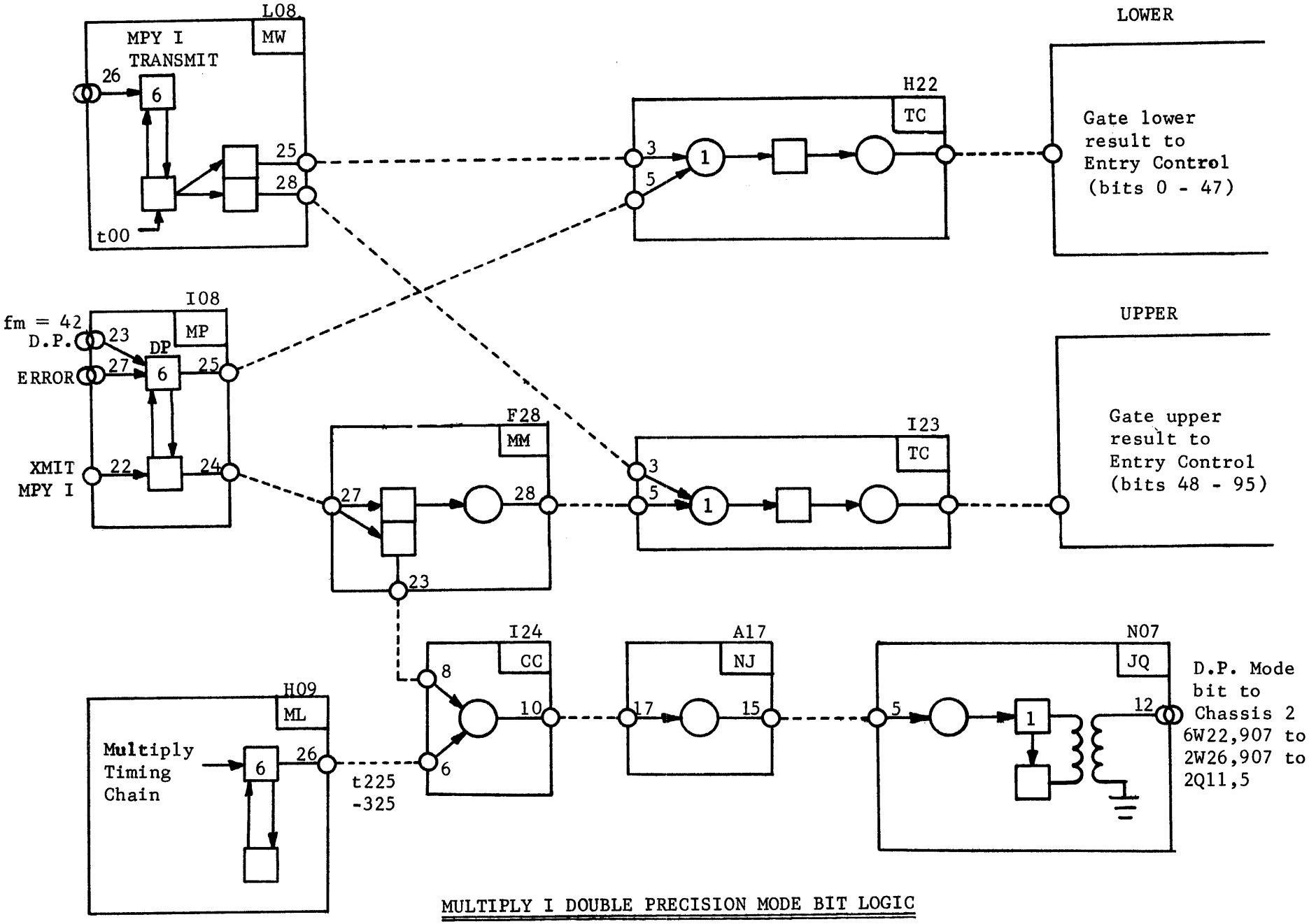
A "DOUBLE PRECISION" mode bit is transmitted to chassis 6 if fm = 42 is translated from the U2 Register (Figure 7.5-3). It is caught on module I08, ANDed with the Multiply I "Transmit" signal on H22, and fanned out to enable the lower bits (0 - 47) of the 96-bit product to Register Entry Control. The absence of the Double Precision mode bit enables bits 48 - 95 of the final product to Entry Control upon receipt of the "Transmit" signal (I23).

GATING OF THE MULTIPLY ROUND MODE BIT TO THE ADDER



t325 - 425

(first iteration)



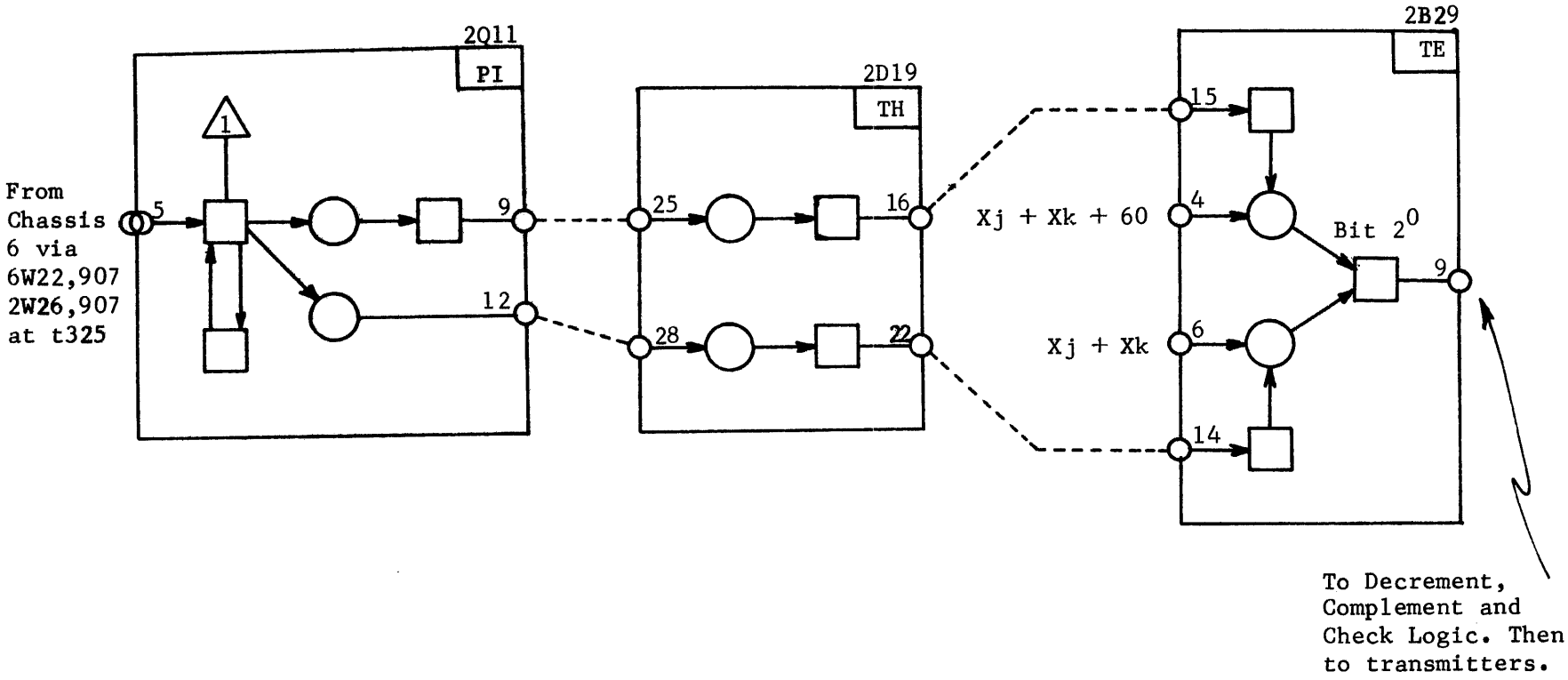
MULTIPLY I DOUBLE PRECISION MODE BIT LOGIC

Figure 7.5-3

The selection of the single or double precision exponent is made by transmitting the Double Precision bit to chassis 2 at approximately time 300 of the Multiply timing chain (the t00 following Multiply time 225 - 325) via cable W22, 907 (Figure 7.5-3). On chassis 2, the single or double precision exponent is selected and gated to the following places for further adjustment and manipulation (See Sections 7.5-10 through 7.5.13 for the more detailed analysis of the exponent logic). Refer to figure 7.5-4.

1) Decrement Logic - used if coefficient overflow is sensed.
2) Complement Logic - packs the true or false value of the exponent, depending upon the final coefficient sign.
3) Error Check Logic - checks for exponent overflow

Note that the Double Precision flip-flop may also be set, via pin 27, if an error condition (zero, indefinite or infinite operand) exists or is generated on chassis 2. For any of the three error possibilities, the coefficient of the result must be cleared. This is accomplished by forcing zeros out of the lower 48-bits of the 96-bit result and setting the Double Precision flip-flop (Figure 7.5-5) By this process, the upper 48-bits need not be disabled, since for either single or double precision the lower 48-bits are selected.



DOUBLE PRECISION LOGIC, CHASSIS 2

Figure 7.5-4

FANOUT OF TRANSMIT - DOUBLE PRECISION

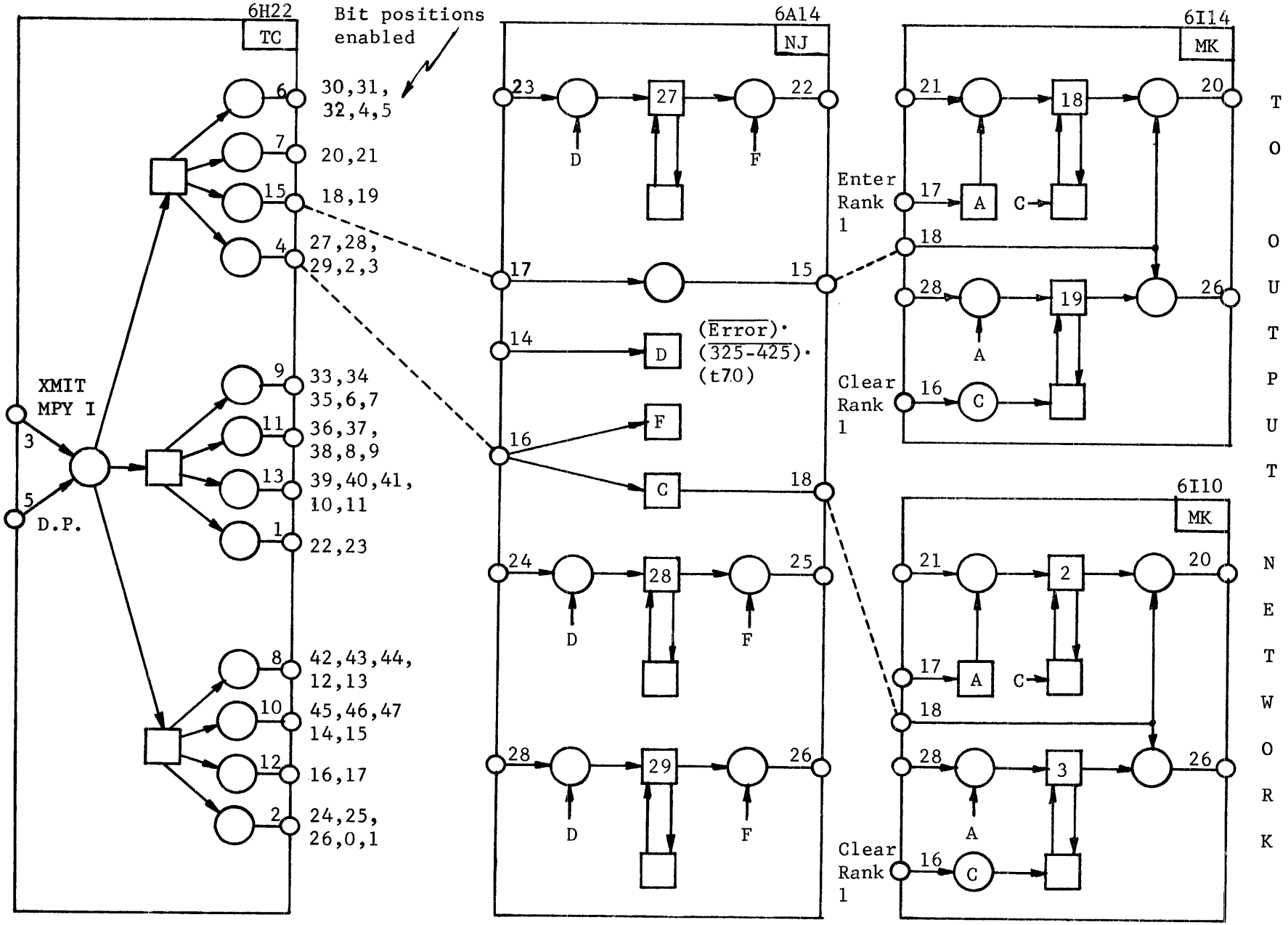


Figure 7.5-5

final product bits 0-5 (which, after the fourth iteration, is stored in the MK modules) from selecting a multiple of XK and gating it to the 3-level adder. The translation is disabled until after the Merge operation has been completed.

17) Form 3XK - No logic gate exists to enable the formation of 3XK. This term is shown to indicate the time allowed for the generation of 3XK - the time period from entering XK into the feeder registers (MA) until the first iteration begins.

18) Clear Partial Sum/Carry Registers (NJ) - The PS and PC registers are cleared every minor cycle until after the fourth iteration. At this point, the clearing gate is disabled until after the final product has been generated (by the Merge phase) and the "Transmit" has been received from the scoreboard.

19) Set Partial Sum/Carry Registers (NJ) - The first four occurancies of this signal on the timing chart (t100 through 300) are meaningless since the purpose of the signal is to gate the output of the 3-level adders into the PS and PC registers, and until t300 no meaningful inputs are fed to the adders. From t300 to t400, setting PS and PC is disabled but because the clear pulse (term 18) does occur, the PS and PC inputs to the first level of the adder will be all zeros on the first iteration. From t400 to t800, four setting pulses occur which gate the results of each iteration into PS and PC. Further setting is disabled until completion of the Merge operation and receipt of the "Transmit" signal from the Scoreboard. At that time (t1025) bits 24-80 of the final product are enabled from the

lower 3-level adder into the lower PS and PC registers. Since a full add occurs during Merge, the Partial Carry outputs will be all zero and only the lower PS register will contain meaningful information.

20) Merge PS, PC and Fully Added Bits (NJ and MK) - This gate enables the Merge phase of the Multiply operation during which the following values are combined to form the final product as shown in Figure 7.5-9.

a) Bits 0-23 of MK lower (final).

b) Bits 6-56 of PS lower (from fourth iteration).

c) Bits 6-56 of PC lower (from fourth iteration).

d) Bits 6-23 of MK upper (from fourth iteration).

e) Bits 0-53 of PS upper (from fourth iteration).

f) Bits 0-53 of PC upper (from fourth iteration).

The gate is generated on module H25, TP4, according to the following Boolean formula:

$$(725\text{-}825) + (775\text{-}875) + (825\text{-}925) + (825\text{-GO MPY}) \Rightarrow \text{Merge}$$

or, in simplified form:

$$725 - \text{XMIT} \Rightarrow \text{MERGE}$$

In other words, Merge is enabled at t725 (after the fourth iteration) unitl the receipt of the next "GO MPY" signal from the scoreboard. Merge is discussed briefly in Section 7.5.1 and a detailed analysis appears in Section 7.5.9 so further discussion is not presented at this point.

7.5.4 COEFFICIENT TIMING SEQUENCE

This discussion deals primarily with the cofficient timing sequence on chassis 6. The exponent timing (chassis 2) is handled separately by the logic and is therefore discussed in later sections dealing with exponent manipulation (i.e. 7.5.10). Some signals, which coordinate the exponent sequence with the coefficient sequence, are mentioned briefly to provide a more complete comprehension of the overall timing operation.

The coefficient timing sequence is initiated upon receipt of the "Go Multiply" signal from the scoreboard. The coefficient timing chain is located on modules H09 and H10 (C.E. Diagrams, sheets 141 and 142) and is composed of 16 flip-flops arranged in a chain. Every other flip-flop is clear/set with a t00; alternate flip-flops with a t50. Each is set for one minor cycle, with the exception of the last which remains set until the start of a new sequence. The outputs of these flip-flops, singly and in combination, are used to sequence the manipulations of the Xj and Xk coefficients required to form the 96-bit product.

An explanation of each term on the coefficient timing chart (figure 7.5-7) follows. In conjunction with the timing chart, the C.E. Diagrams and chassis 6 wire tabs should be utilized to further clarify the discussion of coefficient generation. It is emphasised that this discussion deals with the Multiply I unit and that Multiply II is a separate unit, but similar in operation.

1. Scoreboard Issue - This is the time reference for the timing chart - the scoreboard issue of the Multiply opcode.

MULTIPLY F. U. COEFFICIENT TIMING SEQUENCE

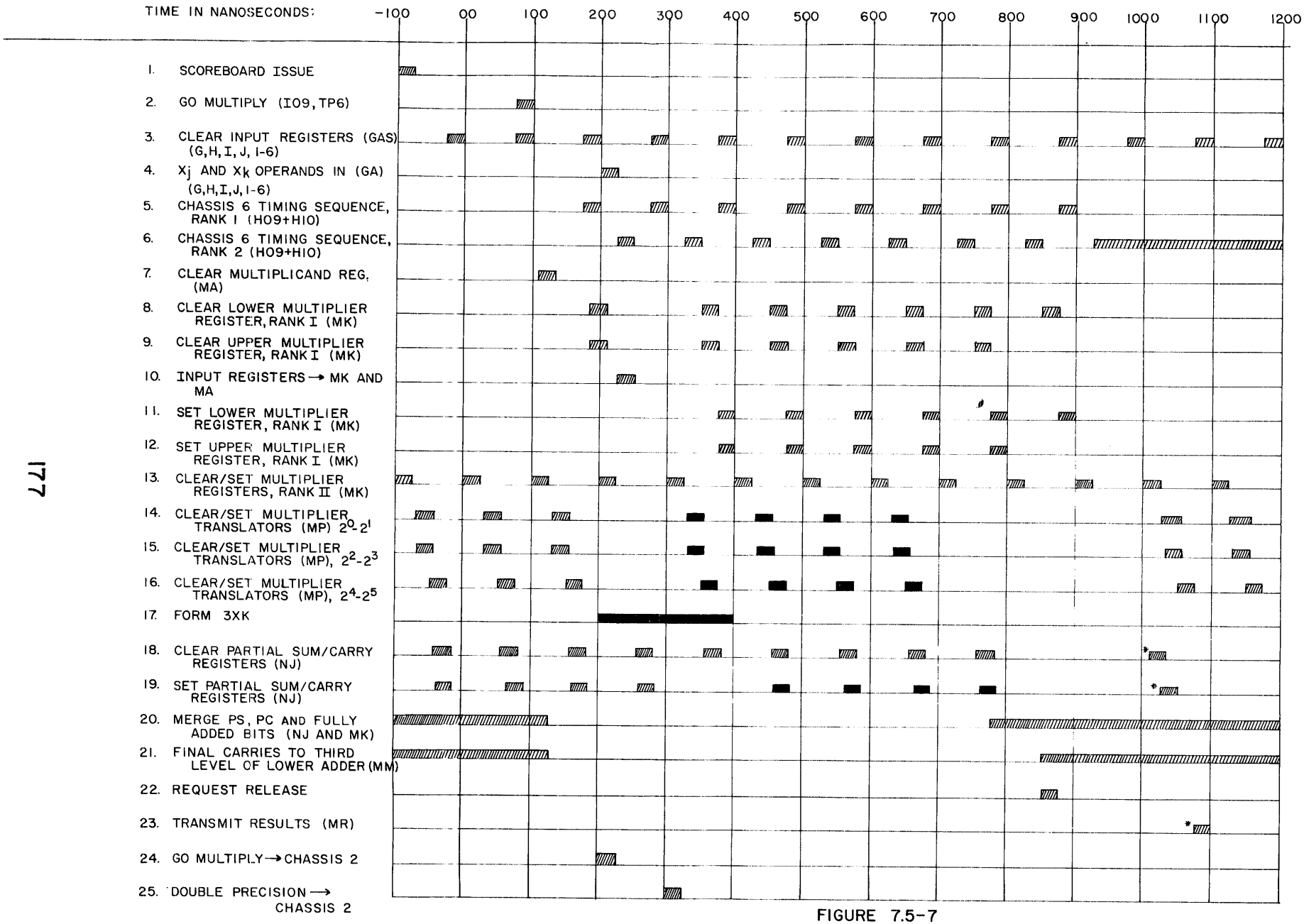


FIGURE 7.5-7

* EARLIEST POSSIBLE TIME — NO THIRD ORDER CONFLICTS

2. Go Multiply I or II - Assuming no second order conflicts, the "Go Multiply" signal will be received on chassis 6 approximately 175 nanoseconds after Scoreboard issue. This signal initiates the Multiply (I or II) timing chain. The "Go Multiply" is gated to chassis 2 approximately 125 nanoseconds later. (See term #24 and figure 7.5-6)

3. Clear Input Registers - The input registers are cleared every minor cycle, as shown. The common "Clear GA" signal can be read from 6G7, TP2 and occurs at approximately t40.

4. Xj and Xk Operands In - The Xj and Xk operands will be received into the chassis 6 input registers (GA modules) at approximately t200.

5. Chassis 6 Timing Sequence, Rank 1 - The timing chain is a string of flip-flops located on modules H09 and H10. The Rank 1 flip-flops are set at t75 and the Rank 2 flip-flops (see term #6) are set at t25. Each flip-flop (except the last in Rank 2) is set for one minor cycle. The chain is initiated upon receipt of the "Go Multiply" signal and is used to sequence the generation of the product of the coefficients.

6. Chassis 6 Timing Sequence, Rank 2 - See term #5.

7. Clear Multiplicand Register - This register is cleared in preparation for the receipt of the Multiplicand (Xk) from the Input Register. (See term #10) It is from this register that 1, 2 and 3 times Xk are formed and selected on the MF and MB modules.

"GO MULTIPLY I" GENERATION AND DISTRIBUTION

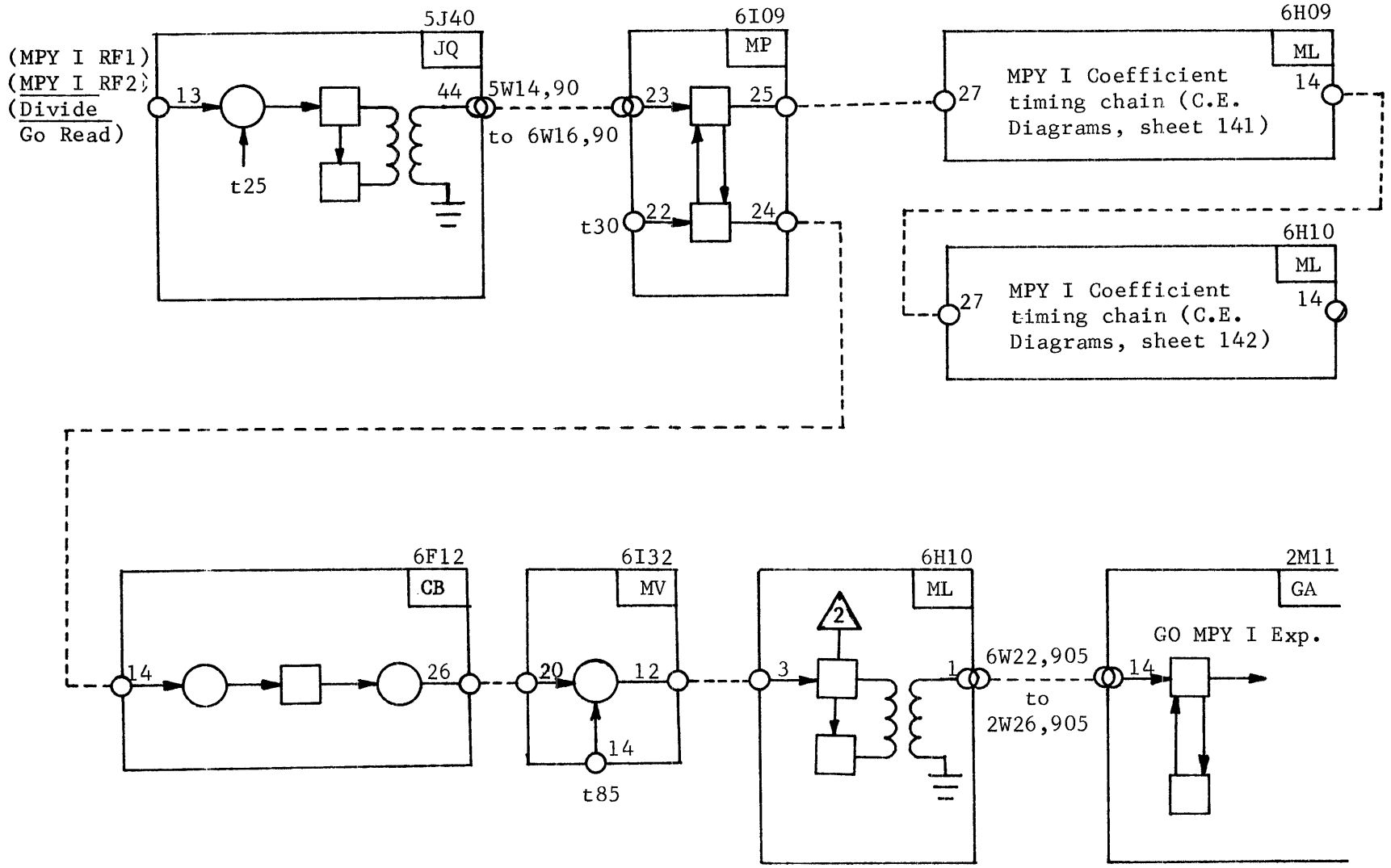


Figure 7.5-6

8 & 9. Clear Upper and Lower Multiplier Register, Rank I -

The first pulse shown (t200) clears the Multiplier register in preparation for the receipt of the Multiplier (Xj) from the Input Register (see term #10). The remaining pulses clear the registers prior to entering the 6 bits of the product from the 6-bit adders (generated during each of 4 iterations) and right shifting the multiplier six places for each iteration (see term #11 and #12). Note that the Upper Multiplier register is not cleared after the last iteration since the output of the upper six bit adder is not used at that time; instead, the Partial sums and carries generated by the fourth upper iteration are used during merge.

Since the Multiplier registers contain two ranks and the second rank copies the first every minor cycle no information is lost by this (see term #13) clearing process.

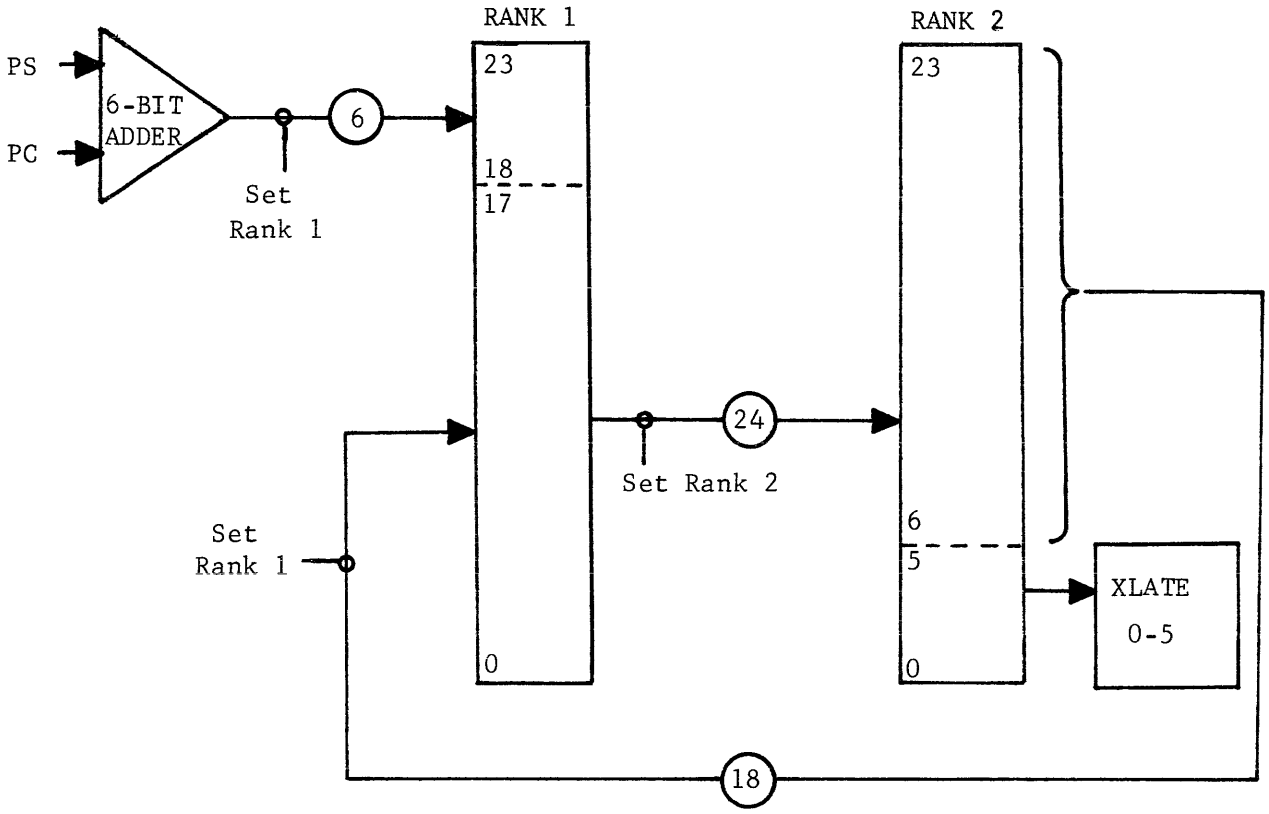
10. Input Registers MK and MA - This pulse enters the two operands into the multiply feeder register located on MK (for Xj) and MA (for Xk) modules.

11 & 12. Set Upper and Lower Multiplier Registers, Rank I -

These pulses result in two transfers:

1) The output of the 6 bit adders is sent to bits 18-23 of the Multiplier registers.

2) Bits 6-23 of the Multiplier registers, Rank 2, are right shifted 6 bit positions into Rank 1.



MULTIPLIER REGISTERS (MK)

Figure 7.5-8

It should be noted that the lower Multiplier Register is loaded a total of seven times (including the initial entry of the multiplier (term #10) and the upper register six times. The reason can be understood by considering the following (see figure 7.5-8)

a) Multiplier translations are made from bits 0-5 of Rank 2 of the upper and lower registers.

b) The multiplier is translated only four times (with meaning) during each iteration (see terms #14, 15 & 16).

c) The outputs of the 6-bit adders are meaningless to this product unless 1) the multiplier is translated, 2) 0, 1, 2 or Xk are selected, and 3) these inputs are filtered through the 3-level.

d) The filter time through the 3-level and 6-bit adders totals about 225 nanoseconds. (i.e. from the time a multiplier translation is made until the result of that iteration is at the output of the 6-bit adder equals 2¼ minor cycles).

Thus in justifying the setting of the Mk modules, terms 14, 15, 16 (set translators) and 19 (set PS and PC registers) of the timing chart must also be considered. The seven entries into the Multiplier registers and the implications of each are outlined below. Keep in mind that Multiplier Rank 2 copies Rank 1 each minor cycle.

First - a) Initial entry of multiplier into Mk modules t225 (term #10)

b) Translate bits 0-5 and store translations at t335 (terms 14, 15, & 16)

Second - a) Set upper and lower Mks at t375 ⇒ 1) right shift bits 6-23 and 2) enter meaningless output of 6-bit adder in bits 18-23 (terms 11 & 12).

b) Translate bits 0-5 and store translations at t435 (terms 14, 15 & 16).

c) Set PS & PC registers and begin first meaningful 6-bit add at t460 (term #19).

Third - a) Set upper and lower MKs at t475 ⇒ 1) right shift bits 6-23 and 2) enter meaningless output of 6-bit adder in bits 18-23 (terms 11 & 12).

b) Translate bits 0-5 and store translations at t545 (terms 14, 15 & 16).

c) Set PS & PC registers and begin second meaningful 6-bit add at t560 (term #19).

Fourth - a) Set upper and lower MKs at t575⇒1) right shift bits 6-23 and 2) enter first meaningful 6-bit add in bits 18-23 (terms 11 & 12).

b) Translate bits 0-5 and store translations at t635 (terms 14, 15 & 16).

c) Set PS & PC registers and begin third meaningful 6-bit add at t660 (term #19).

Fifth - a) Set upper and lower MKs at t675⇒1) right shift bits 6-23 and 2) enter second meaningful 6-bit add in bits 18-23 (terms 11 & 12).

b) Translation of bits 0-5 is disabled at this time since the entire multiplier has been translated (terms 14, 15 & 16).

c) Set PS & PC registers and begin fourth meaningful 6-bit add at t760 (term #19).

Sixth - a) Set upper and lower MKs at t775⇒1) right shift bits 6-23 and 2) enter third meaningful 6-bit add in bits 18-23 (terms 11 & 12).

b) Translation of bits 0-5 is still disabled (terms 14, 15 & 16).

c) Setting of PS & PC registers is disabled at this point since the four passes through the 3-level adders have been completed.

Seventh - a) Set lower MKs at t875⇒1) right shift bits 6-23 and 2) enter fourth meaningful 6-bit add in bits 18-23 (terms 11 & 12). The upper MKs are not set on the fourth iteration since the upper PS & PC contents are used directly during the merge phase.

At this point the status of the Multiply sequence is as follows:

a) The lower MKs contain 24-bits of the final produce in bits 0-23.

b) The upper MKs contain 18 bits of the final product (may be modified by a carry from bit 2^{23}) in bits 6-23.

c) The upper PS and PC registers contain the results of the fourth iteration. The lower six bits were <u>not</u> added, so all bits are meaningful.

d) The lower PS and PC registers contain the results of the fourth iteration. The lower six bits were added so only the upper bits are meaningful.

The above values will be combined during the Merge phase of multiply to generate the final 96-bit product.

13) Clear/Set Multiplier Registers Rank II - This signal occurs each minor cycle at t00 and causes Rank 1 of the multiplier registers (MK modules) to be copied into Rank 2. (See Figure 7.5-8). The selection of 0, 1, 2 or 3XK is made by translating bits 0-5 of Rank 2.

14) Clear/Set Multiplier Translators (MP) - This signal enables storing
15) &
16) the 2-bit translations of the lower six bits of each multiplier in separate flig-flops. The flip-flop locations and the associated meaning is shown in the following table:

	LOWER			UPPER		
Bit:	$2^0,2^1$	$2^2,2^3$	$2^4,2^5$	$2^{24},2^{25}$	$2^{26},2^{27}$	$2^{28},2^{29}$
1X	I7,TP4	I9,TP4	I8,TP4	I7,p15	I9,p15	I8,p15
2X	I7,TP2	I9,TP2	I8,TP2	I7,p19	I9,p19	I8,p19
3X	I7,TP3	I9,TP3	I8,TP3	I7,p18	I9,p18	I8,p18

The outputs of these flip-flops will be used to select the proper multiple of the multiplicand (0X, 1X, 2X or 3X) to be fed to each level of the 3-level adder during each iteration of the multiply sequence. Note that only four of the enables (t300-700) are meaningful to the multiply step in process. Disabling the translation during the minor cycle, t200-300, selects 0X for all initial inputs to the 3-level adder. This causes the partial sums and carries to be all zero for the first iteration. Disabling the translation after the fourth iteration (t700-1000) prevents the

21) Final Carries To Third Level Of Lower Adder (MM) - During the Merge phase of multiply, a full add must take place to generate the final product. Since the lower 3-level adder is used to generate 57 bits (24-80) of the product, a special logic circuit exists which converts the third level of the lower Added into a full adder. This is the "Merge Carry Network" which is used only during Merge; not during the normal four iterations.

The gate which enables the Carry Network is shown on the timing chart (term 21). Note that it is disabled during normal iterations (t125-850) and enabled at all other times. The term is translated (6F9, TP4) as follows:

(825-925) + (925-GO MPY) $\Rightarrow$ Enable Final Carries

or, in simplified form:

t825 - GO MPY

The gate is thus enabled from t825 until receipt of the next GO MPY from the scoreboard. See Section 7.5.9 for the detailed analysis of the Merge phase.

22) Request Release - The Request Release is transmitted from chassis 6 (H09, TP2) to the scoreboard at t850 of the Multiply timing sequence. This signal resolves any third order conflicts which may exist.

23) Transmit Results (MR) - Assuming that no third order conflicts exist, the result will be transmitted from the MR modules at t1075 of the Multiply sequence. A Double Precision result (bits 0-47) is trans-

mitted if the D.P. mode is high (Figure 7.5-5); a Single Precision result is sent if D.P. is low.

24) Go Multiply→Chassis 2 - This signal initiates the exponent timing chain on chassis 2. It is transmitted from chassis 6 (6H10, TP2) at t200 (see Figure 7.5-6).

25) Double Precision→Chassis 2 - This signal is sent to chassis 2 (N07, TP1) at t300 to enable the selection of the sum of the exponents (if D.P. mode). If the mode bit is not sent to chassis 2, the sum of the exponents - $60_{(8)}$ (S.P. mode) will be selected as the final exponent. See Figures 7.5-3 and 7.5-4.

7.5.5 1, 2, 3 TIMES MULTIPLICAND (XK)

GENERAL:

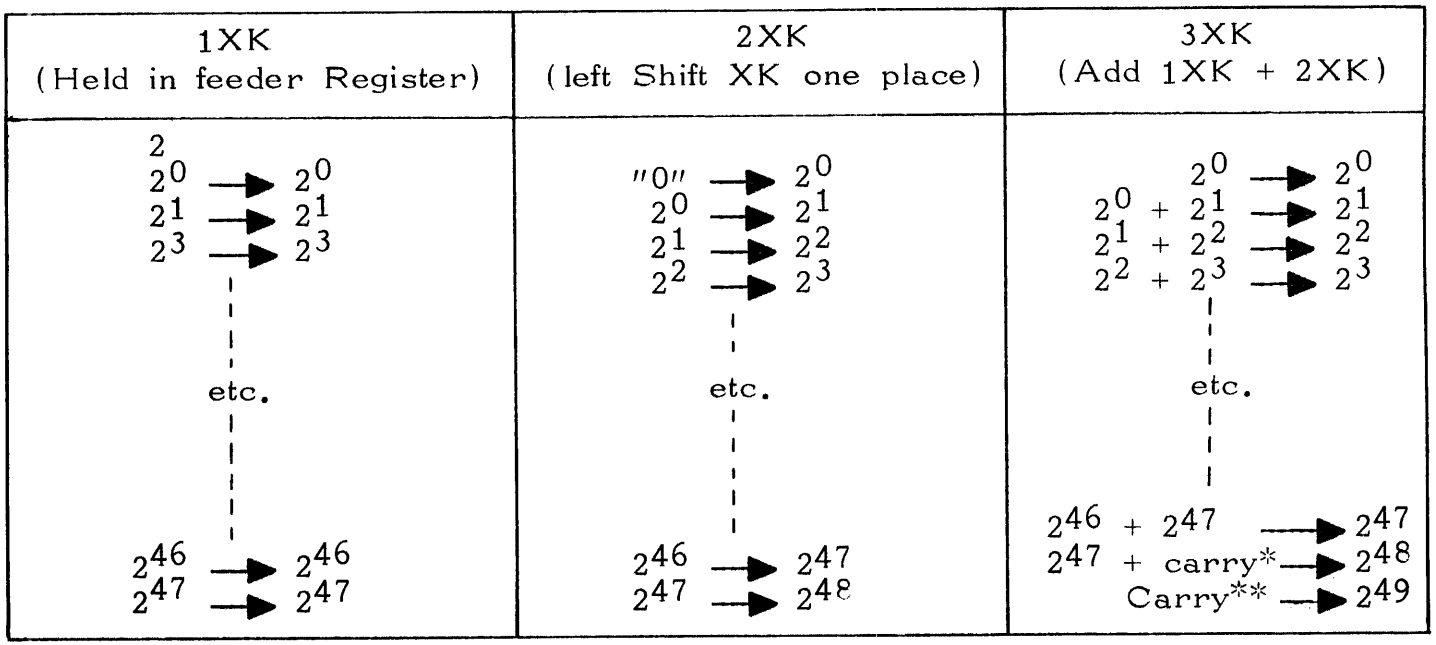
The function of the logic discussed in this section is two fold:

1) The formation of 1, 2, and 3 times the multiplicand (XK).

2) The distribution of the selected multiple of XK to the three levels of both the Upper and Lower 3-Level Adders as specified by each pick (2-bits of the multiplier).

At the beginning of the multiply operation, the multiplicand (XK) is placed in its holding register located on MA modules. It remains there, unattended, throughout the multiply step. This register feeds a static network which forms 1, 2 and 3 times XK. Since 1XK exists in the holding register, no additional logic is required for its formation. 2XK is formed by left shifting the multiplicand 1 bit position, thereby multiplying by two. 3XK is formed with a special full adder which adds 1XK and 2XK. Due to the propagation of carries to high order bits, the value, 3XK, may be 50 bits in length. (See Figure 7.5-9.)

The 3 multiples of XK must be distributed to the Upper and Lower adders at four different times (once per iteration) during the Multiply operation. (See terms 14-16 of Figure 7.5-7.) This selection and distribution is accomplished by translating the lower six bits of the Upper (2^{24}-2^{29}) and Lower (2^{0}-2^{5}) Multiplier Registers. Six translations (of two bits each) occur during each iteration and gate the proper XK multiple to each level of the three level adders. In the event that the two bits translated are zeros, neither XK, 2XK nor 3XK is gated to the adder level and all zeros are therefore



* Possible carry from stage 47
** Possible carry from stage 48

REQUIREMENTS FOR FORMATION OF 1, 2, AND 3XK

Figure 7.5-9

FORMATION OF
1, 2, and 3 XK

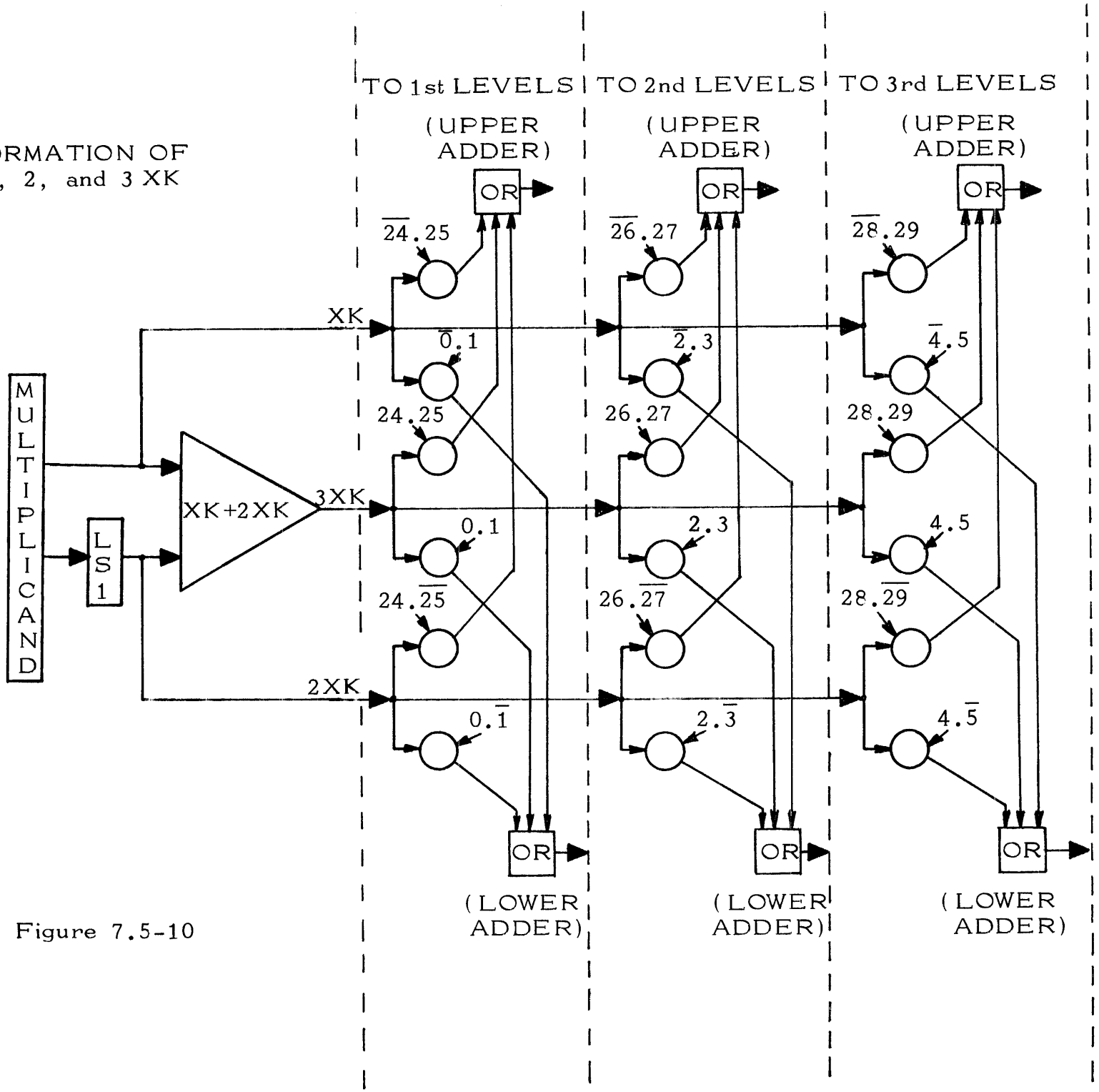


Figure 7.5-10

entered. Figure 7.5-10 illustrates, in block diagram form, the formation, selection, and distribution of 1, 2 and 3XK.

LOGIC ANALYSIS:

Figure 7.5-11 should be used in following the logic analysis of the 1, 2 and 3XK circuitry. The 3XK adder is not shown completely (the carry determination and propagation logic is missing) so reference to the Chassis 6 wire tabs should also be made.

Selection and distribution of the value, 1XK, requires that the multiplicand be gated from the feeder register (MA modules) to the desired level(s) of the 3-level adder(s), via the MB modules. Bit 0, for example, is distributed from C01, pin 12 ($\overline{XK2^0}$) to A18, pin 12. On A18, term F is a "one" if Bit 2^0 of XK is a "one". At the bottom of A18 five terms, A,B,C,D and E, are used to select XK, 2XK or 3XK for each pick of the multiply operation. Term A, for example, indicates that the bit 0 and 1 configuration of the multiplier is $01_{(2)}$. This, ANDed with term F, gates bit 2^0 of XK to bit 2^0 of the first level of the Lower 3-level adder. Term E specifies that bits 24 and 25 of the multiplier equal $01_{(2)}$. This term, ANDed with term F, enables bit 2^0 of XK to bit 2^0 of the first level of the Upper 3-level adder. Terms A and E also gate term G (a "one" indicates that bit 2^1 equals a "one") to bit 2^1 of the Lower (pin 22) and Upper (pin 8) 3-level adders, and term H (XK bit 2^2) to bit 2 of the adders.

The 0 or 1 state of XK bit 2^0 is gated from A18, pin 7 ($\overline{XK2^0}$) to

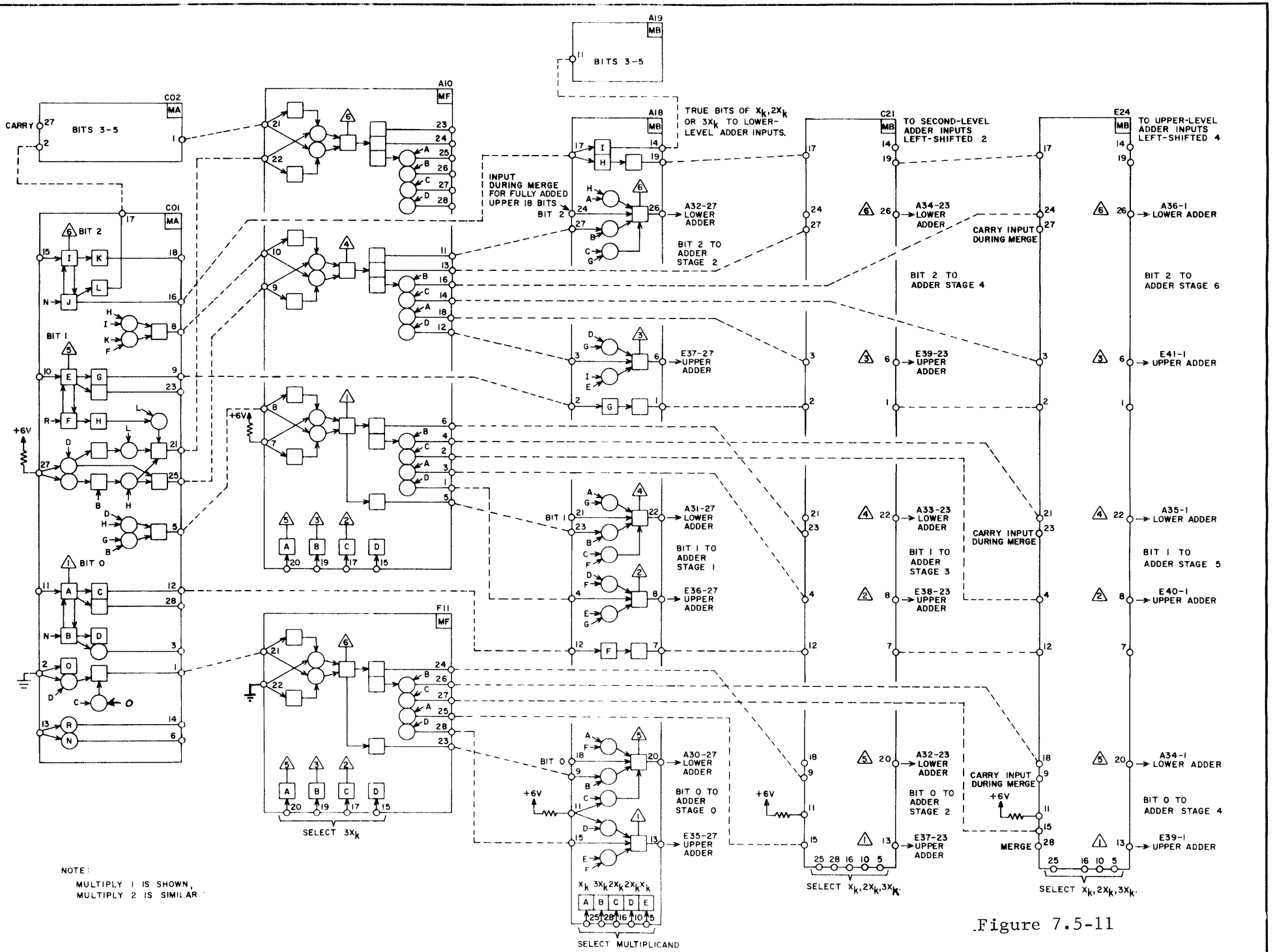

A19 MB
BITS 3-5
A10 MF
C02 MA
CARRY
BITS 3-5
TRUE BITS OF X_k, $2X_k$ OR $3X_k$ TO LOWER-LEVEL ADDER INPUTS.
A18 MB
C21 MB
TO SECOND-LEVEL ADDER INPUTS LEFT-SHIFTED 2
E24 MB
TO UPPER-LEVEL ADDER INPUTS LEFT-SHIFTED 4
INPUT DURING MERGE FOR FULLY ADDED UPPER 18 BITS
C01 MA
BIT 2
A32-27 LOWER ADDER
BIT 2 TO ADDER STAGE 2
A34-23 LOWER ADDER
CARRY INPUT DURING MERGE
A36-1 LOWER ADDER
BIT 2 TO ADDER STAGE 4
BIT 2 TO ADDER STAGE 6
BIT 1
E37-27 UPPER ADDER
E39-23 UPPER ADDER
E41-1 UPPER ADDER
+6V
A31-27 LOWER ADDER
BIT 1 TO ADDER STAGE 1
A33-23 LOWER ADDER
BIT 1 TO ADDER STAGE 3
A35-1 LOWER ADDER
BIT 1 TO ADDER STAGE 5
BIT 0
E36-27 UPPER ADDER
E38-23 UPPER ADDER
E40-1 UPPER ADDER
F11 MF
A30-27 LOWER ADDER
BIT 0 TO ADDER STAGE 0
A32-23 LOWER ADDER
BIT 0 TO ADDER STAGE 2
A34-1 LOWER ADDER
BIT 0 TO ADDER STAGE 4
E35-27 UPPER ADDER
E37-23 UPPER ADDER
E39-1 UPPER ADDER
SELECT $3X_k$
SELECT $X_k, 2X_k, 3X_k$
SELECT MULTIPLICAND
NOTE:
MULTIPLY 1 IS SHOWN,
MULTIPLY 2 IS SIMILAR.

Figure 7.5-11

C21 pin 12. C21 is another MB module that distributes bits 0-2 of the selected XK multiple to the second levels of the Upper and Lower adders. On this module, terms A, B, C, D and E are translations of bits 2 & 3 and 26 & 27 of the multiplier. Term A, for example, indicates that bits 2 & 3 of the multiplier equal $01_{(2)}$ and will gate bits 2^0, 2^1 and 2^2 of XK to the second level adder inputs. Since this pick is 2 bits more significant than the first, the value XK will be fed to the second level left shifted 2 places (i.e. bit 0 feeds bit 2, bit 1 feeds bit 3, etc.).

Bits 0-2 of the third pick are distributed in a similar manner on E24, also an MB module. The select gates, A, B, C, D and E in this case translate bits 4 & 5 and 28 & 29 of the multiplier to enable pick #3 to the third levels of the Upper and Lower adders. Since this selection is 4 bits more significant than the first pick, the XK multiple is fed to the third levels left shifted 4 places (bit 0 feeds bit 4, bit 1 feeds bit 5, etc.).

Terms C and D on the MB modules translate each 2 bit pick for a $10_{(2)}$ configuration to enable the value 2XK to the designated levels of the 3-level adders. Note that on module A18, the 2^0 output is forced to a "zero" (pin 11 is a constant 1.2v). This occurs, since in forming 2XK, XK is left shifted one place and the 2^0 input to the adder is consequently always a "zero". (Figure 7.5-9.) For the remaining bits of the value 2XK, each bit of the XK feeder is gated to the next significant bit position. On the MB modules, for example, bit 2^0 (represented by term F) is enabled to bit 1 of the

first level of the Lower adder if term C (which indicates a $10_{(2)}$ configuration in bits 0 and 1) is a "one". Term D, which translates bits 24 and 25 for $10_{(2)}$, enables bit 0 of XK to bit 1 of the first level.

In distributing 2XK to the second levels, the value is left shifted two places with respect to the first level inputs. In other words, bit 2^0 of XK (MA modules) is fed to bit 2^3 of the second levels, bit 2^1 to 2^4, etc. The third level inputs are left shifted 2 more places, so bit 2^0 of XK feeds 2^5, 2^1 feeds 2^6, etc.

Distribution of the value, 3XK, differs somewhat from that described for XK and 2XK. The output network of the 3XK adder (described later) is located on MF modules. Note that six outputs for each bit position are shown - that is, one for each level of each adder. Bit zero of the sum, for example, is gated from F11, pins 23, 24, 25, 26, 27 and 28. The following table shows the destination for each output of bit 2^0, and can be proven by studying Figure 7.5-11 and the Chassis 6 wire tabs.

PIN	ADDER	LEVEL	GATING TERM
23	Lower	one	A18,B
24	Lower	two	C21,B
25	Upper	two	F11,A
26	Lower	three	F11,B
27	Upper	three	F11,C
28	Upper	one	F11,D

Table 7.5-1

Pins 23 and 24 represent the true value of the sum; pins 25 through 28 the false value. The four select gates A, B, C and D, on the MF modules are used to gate 3XK to all three levels of the Upper adder and to Level #3 of the Lower adder. Gating to Levels 1 and 2 of the Lower adder takes place on the MB modules via Term B (i.e. A18 and C21 for bit 2^0). Table 7.5-1 summarizes the modules and terms used to gate bit 2^0 of 3XK to the six levels, and is representative of all bit positions of 3XK. Note that gating 3XK bit 2^0 to the third level of the Lower adder occurs via E24 pin 18 whereas the first and second level inputs are gated via pins 9 of A18 and C21. This occurs, since during the Merge phase carry inputs from the Merge Carry Network enter level three of the Lower adder via pins 9, 23 and 27 of the MB modules.

The above description of XK, 2XK and 3XK selection and distribution dealt primarily with bits 0-2, since the logic for those bits is shown in Figure 7.5-11. The remaining bits of the multiplicand are distributed similarly (in 3 bit groups) by the MF and MB modules. Table 7.5-2 shows the MB modules used to select and distribute each three bit groups. The wire tabs should be used for the specific wiring of the distribution logic.

3XK ADDER:

The 3XK Adder is a full adder which forms the sum of XK and 2XK. Since 2XK is formed by left shifting XK one bit position, only one feeder register is required. Each bit of the feeder register (MA modules) has two outputs to the adder logic. One output is to the

MB MODULES

BITS	FIRST LEVELS	SECOND LEVELS	THIRD LEVELS
0-2	A18	C21	E24
3-5	A19	C22	F18
6-8	A20	C23	F19
9-11	A21	C24	F20
12-14	A22	D18	F21
15-17	A23	D19	F22
18-20	A24	D20	F23
21-23	B18	D21	F24
24-26	B19	D22	G18
27-29	B20	D23	G19
30-32	B21	D24	G20
33-35	B22	E18	G21
36-38	B23	E19	G22
39-41	B24	E20	G23
42-44	C18	E21	G24
45-47	C19	E22	H18
48-50	C20	E23	H19

DISTRIBUTION OF XK, 2XK & 3XK

Table 7.5-2

corresponding stage of the adder; the second is to the next significant bit position, as follows:

```
 XK bits          47 46 45 ---------- 3 2 1 0
2XK bits       47 46 45 44 ---------- 2 1 0
3XK bits    49 48 47 46 45 ---------- 3 2 1 0
```

As can be seen, the result may be 50 bits in length because of the possible carry into bit 2^{49}. Note also, that bit 2^0 of the result will always be the same as bit 2^0 of XK since nothing is added to that bit position.

A true addition is performed by the adder (as opposed to subtracting to add, complementing operands, etc.). As a result, the conditions generate, satisfy, enable and pass are defined as follows:

	Generate	Satisfy	Enable
XK	1	0	1 or 0
2XK	1	0	0 or 1

The condition $\overline{\text{Satisfy}}$ is also referred to as a Pass.

The addition is performed in three basic logical steps:

1) Determine whether or not Equivalence exists between the two source operands.

   i.e. EQUIVALENCE $\Rightarrow$ $\begin{matrix}1\\1\end{matrix}$ or $\begin{matrix}0\\0\end{matrix}$

2) Determine into which stages carries are entered. A carry will enter a stage if the previous stage is a generate or if some other less significant stage is a generate and none of the more significant stages are satisfies.

3) Toggle the results of the equivalence checks for all stages that do not have a carry in.

The following example illustrates the preceding steps. The original value of XK is assumed to be $0\text{———}\ 01234567_{(8)}$.

NOTE: 1) S, E, and G refer to Satisfy, Enable, and Generate respectively.

2) Asterisks indicate those stages with a carry in.

				*	**	**	***	***	*
	SS———	S	SEE	EES	EGG	ESE	EEG	GEE	GGE
XK =	0———	0	001	010	011	100	101	110	111(2)
2XK =	00———	0	010	100	111	001	011	101	110
STEP 1 (Equivalence) =	11———	1	100	001	011	010	001	100	110
STEP 2 (Carry In) =	00———	0	000	001	110	011	111	111	100
STEP 3 =	00———	0	011	111	010	110	001	100	101
IN OCTAL =	0———	0	3	7	2	6	1	4	5

Using octal arithmetic, the same result is obtained:

$$\begin{aligned} XK &= 0\text{———}\ 01234567 \\ 2XK &= 0\text{———}\ 02471356 \\ 3XK &= 0\text{———}\ 03726145 \end{aligned}$$

The following formulas which define the sum as a "one" or a "zero" can be derived from the above procedure:

$$\text{Sum} = \text{"1"} \text{ if } \text{Equivalence} \cdot \text{Carry} + \overline{\text{Equivalence}} \cdot \overline{\text{Carry}}$$

$$\text{Sum} = \text{"0"} \text{ if } \overline{\text{Equivalence}} \cdot \text{Carry} + \text{Equivalence} \cdot \overline{\text{Carry}}$$

Several module types are used in the 3XK adder. Their functions are generally described as follows:

MA - Contain feeder register, make initial equivalence check, and propagate carries within a 3-bit group (bits 0-2, 3-5, 6-8, etc.).

ME - Determine the pass and carry-out conditions for 3-bit groups.

MH - Determine carry-outs for six sections (bits 0-9, 10-18, 19-27, 28-36, 37-45, 46-47). Also, final carries into stages are determined for some bits.

MI - Sum up the group and section Pass conditions.

MJ & MH - Make carry propagation checks for carries into 3-bit groups.

MF - Perform final equivalence and carry summation and produce the final sum.

A portion of the 3XK adder logic is shown in Figure 7.5-11, but the carry and pass summation logic is left out. The Chassis 6 wire tabs should therefore be used to understand the adder logic completely.

The equivalence check, which is made on the MA modules, is a simple matter of checking the inputs to each adder stage for a 0/0 or 1/1 configuration. The logic is shown for stages 1 and 2 in Figure 7.5-11. (The check need not be made for bit 2^0 since 2^0 of the result will always be the same as bit 2^0 of XK.) Pin 5, for example, translates as:

$$2^0 \cdot 2^1 + \overline{2^0} \cdot \overline{2^1}$$

or, Equivalence in stage 2^1

Pin 8 translates as:

$$2^1 \cdot 2^2 + \overline{2^1} \cdot \overline{2^2}$$

or, Equivalence in stage 2^2

This check occurs for all bit positions on the MA modules, which are located on C1 through C9 (bits 0-26 of XK) and D1 through D8 (bits 27-50 of XK; bits 48-50 are not used). The result of the equivalence check are sent to the MF modules and are ANDed with the carry-in conditions to generate the final product.

The first step in performing the checks for carries into the stages is to determine which 3-bit groups have carries out (generates) and which have all passes ($\overline{\text{satisfies}}$). This is accomplished on ME modules, each of which is capable of determining the carry out and pass conditions for 3 groups. With 48 bit operands, 16 groups exist. The lower 15 groups contain 3-bits each and the 16th contains 2 bits.

Group 3, bits 7-9, will be used as an illustration which is representative of all groups. On module 6A1 (wire tabs) test point #5 indicates that all 3 stages in group 3 are passes. The translation for TP5 is:

$$2^6 \cdot 2^8 + 2^7 \cdot 2^9 = 1$$

The inputs for group 3 look as follows:

$2^9\ 2^8\ 2^7$
$2^8\ 2^7\ 2^6$

It can be seen by trying all possibilities that the above formula does imply that no satisfies (0/0 configuration) are present.

Test Point #6 on 6A1 indicates that a carry will leave group 3 and will enter group 4. TP6 translates as follows:

$$2^8 \cdot 2^9 + 2^7 \cdot 2^8 + 2^6 \cdot 2^7 \cdot 2^9$$

Again, by studying the inputs for group #3, the formula covers all possibilities of generating a carry <u>from within</u> group 3. (It does not consider that a carry may enter stage 7 from group 2; this is determined later.)

The following table shows the test points and ME module locations associated with the pass and carry checks for each group.

GROUP	CARRY		PASS	
	MODULE	TP	MODULE	TP
0-3	6A1	1	6A1	2
4-6	6A1	3	6A1	4
7-9	6A1	6	6A1	5
10-12	6A2	1	6A2	2
13-15	6A2	3	6A2	4
16-18	6A2	6	6A2	5
19-21	6A3	1	6A3	2
22-24	6A3	3	6A3	4
25-27	6A3	6	6A3	5
28-30	6B1	1	6B1	2
31-33	6B1	3	6B1	4
34-36	6B1	6	6B1	5
37-39	6B2	1	6B2	2
40-42	6B2	3	6B2	4
43-45	6B2	6	6B2	5
46,47	6B3	1	6B3	2

The MH modules further combine the group pass and carry conditions to determine which sections have carries out. For example, 6B5,TP2 (pin 9) indicates that Section 1 (bits 0-9) generates a carry out. The translation for TP2 is:

7-9 Carry out + 7-9 Pass · 4-6 Carry out + 7-9 Pass · 4-6 Pass · 0-3 Carry out

If the conditions for the above formula are met, section 0-9 has a carry-out. The following table shows where section carry-outs may be checked by Test Point:

SECTION	MODULE	TEST POINT
0-9	6B5	2
10-18	6B5	5
19-27	6B6	2
28-36	6B6	5
37-45	6B7	2
46-47	6B7	6

The MH modules also combine the group carry-in conditions and gate the carry-in for a given group to pin 7 of the MF module for that group. On Figure 7.5-11, this pin is always a "zero" since a carry into bit 1 is not possible (no end-around-carry occurs). On the MFs of the remaining groups, pin 7 will be tied to an MH module. For example, 6B5, p22, where a "one" indicates a carry-in condition to stage 13 (the lowest bit of the fifth group). Note that pin 22 is connected to B11, pin 7 which is the final carry input to bit 13. Pin 24 of 6B5 is wired to pin 27 of C5, an MA module. Note from Figure 7.5-11 that pin 27 is the carry input to a group (always "zero" for group 1) and the carry is propagated within a group on the MA module itself (i.e. pins 25 and 21 indicate final carries into the respective bit position).

The MI modules (i.e. 6B4) combine the group pass conditions, determined by the ME modules, into larger section pass conditions. For example, on 6B4,

$\overline{TP1} \Rightarrow$ 46,47,0-9 are passes.
$\overline{TP2} \Rightarrow$ 46,47 are passes.
$\overline{TP3} \Rightarrow$ 37-47 are passes.
$\overline{TP4} \Rightarrow$ 28-36 are passes.
$\overline{TP5} \Rightarrow$ 37-45 are passes.
$\overline{TP6} \Rightarrow$ 28-45 are passes.

These outputs are used by the MJ modules to propagate carries into groups. For example, by translating the inputs to 6A5, the following output translations are obtained.

pins 2 & 15 $\Rightarrow \overline{\text{Carry into 10}}$
pin 14 $\Rightarrow \overline{\text{Carry into 13}}$
pin 1 $\Rightarrow \overline{\text{Carry into 16}}$
pin 28 $\Rightarrow \overline{\text{Carry into 7}}$
pin 26 $\Rightarrow \overline{\text{Carry into 4}}$

The other MJ modules will propagate carries into the remaining groups.

The final sum is generated on the MF modules (see Figure 7.5-11) with an equivalence circuit. For example, 6A10, TP4 is the equivalence circuit for stage 2^0 of the adder. If the two inputs (pins 10 and 9 which are the stage equivalence and carry-in conditions respectively) are both ones or both zeros, the output on pin 11 will be a "one". The outputs of the test points (i.e. #4) are actually the false value of the sum, 3XK. This value is now gated through the 1, 2 and 3XK selection and distribution circuits which are discussed earlier in this section.

Since a relatively comprehensive understanding of other 6600 adders is assumed, further discussion of the 3XK Adder is felt to be somewhat wasteful. At this point, the concept of the adder should be quite clear. If further analysis is desired, this concept should guide the research of the Chassis 6 wire tabs.

7.5.6 THREE LEVEL ADDERS

Recall that a 6600 Multiply unit examines two bits of the multiplier at a time and uses these two bits to select one of the four quantities, 0X, 1X, 2X or 3X, where X represents the multiplicand. This quantity is put into the Three Level Adder along with the results of two other 2-bit picks and the result of the previous iteration. Thus, it multiplies by six bits per iteration. In four iterations, it can multiply by 24 bits. Since the upper and lower 24-bits are multiplied separately, the upper and lower products are obtained in four iterations. The final step is the addition of the upper and lower products to obtain the final answer. It was also noted that the lowest 6-bits of the answer after each iteration need not be put back into the adder during the next iteration. In fact, the lowest 24-bits of the lower product need not be put back into the adder during merge since these are 24-bits of the final answer. However, the lowest 24-bits of the upper product must be sent back to the adder during merge since they must be added to the upper 24-bits of the lower product.

The following example is used in discussing the Three Level Adders:

```
                    0 1 2 3 4 5 6 7
                    1 1 1 1 1 1 1 1
                    0 1 2 3 4 5 6 7
                  0 1 2 3 4 5 6 7
                  0 1 3 6 0 2 4 5 7     (FIRST Iteration)
                0 1 2 3 4 5 6 7
              0 1 2 3 4 5 6 7
              0 1 3 7 4 0 5 0 3 5 7     (SECOND Iteration)
            0 1 2 3 4 5 6 7
          0 1 2 3 4 5 6 7
          0 1 3 7 4 2 0 6 4 0 3 5 7     (THIRD Iteration)
        0 1 2 3 4 5 6 7
      0 1 2 3 4 5 6 7
      0 1 3 7 4 2 0 7 7 6 4 0 3 5 7     (FOURTH Iteration)
```

Figure 7.5-12

In the case of the 6600 Multiply units, the adders must be able to add four quantities together. These are the results of each of the three picks and the result from the previous iteration. The following example illustrates this using the same operands as in Figure 7.5-12 but written in binary form.

```
                                        000 001 010 011 100 101 110 111
                                        001 001 001 001 001 001 001 001
                                        ---------------------------------
                                        000 001 010 011 100 101 110 111
                                    00  001 010 011 100 101 110 111 0
                                  0 000 000 000 000 000 000 000 00
                                    ---------------------------------
First Iteration                   0 000 001 011 110 000 010 100 101 111
Pick 1 (Xk)                   000 001 010 011 100 101 110 111
Pick 2 (2Xk)                00 001 010 011 100 101 110 111 0
Pick 3 (0Xk)              0 000 000 000 000 000 000 000 00
                          ---------------------------------
                          0 000 011 011 111 100 000 101 000 011 101 111
                        0   0   1   3   7   4   0   5   0   3   5   7   (8)
```

Figure 7.5-13

In this example the $2^0 2^1$ pick was a one so the multiplicand was used as is. The second pick, $2^2 2^3$ pick is a two. Since 2x is equal to the multiplicand shifted left 1 place and since this is the second pick of two bits each, the multiplicand has been shifted left 3 places before addition. The $2^4 2^5$ pick is zero. The sum of these three quantities is the result of the first iteration and it is the same as the result of the first iteration in Figure 7.5-12. The second iteration is the same as the first except that the result of the first iteration must be added in with the next three picks.

It has been stated that the adder is able to add four numbers together during each iteration. This is somewhat of a simplication. The adder could add these numbers and get a complete answer, but it

would require the use of a carry network to resolve all the carries. This would require approximately twice the time than the method actually used. In the 6600 method the sum after each iteration is obtained in two parts called a partial sum and a partial carry. The sum of these two numbers is the actual answer, however the sum is not taken until final merge. Instead, both quantities are put back into the adder for the next iteration.

In order to understand this, the problems involved in adding four numbers together will be analyzed. Assume that somewhere during one of the four iterations for six bits, the result of the previous iteration and the result of each of the three picks are all ones. In this case the adder must not only add the four numbers together, but it must add up to three carries also. In some cases it can accomodate these carries, but in other cases it cannot. When it is not able to propagate the carry during an iteration, a partial carry is generated.

For clarity, the add is broken up into a series of smaller additions of either two or three numbers at a time. The first two numbers to be added are the result of the first pick and the partial sum from the previous iteration. (Of course the partial sum and partial carry will always be zero during the first iteration.) For this example the same two numbers that were used in Figure 7.5-12 are used. The first pick is 1 times the multiplicand (2^0 & 2^1 determine the first pick). The partial sum is zero since this is the first iteration. In binary form:

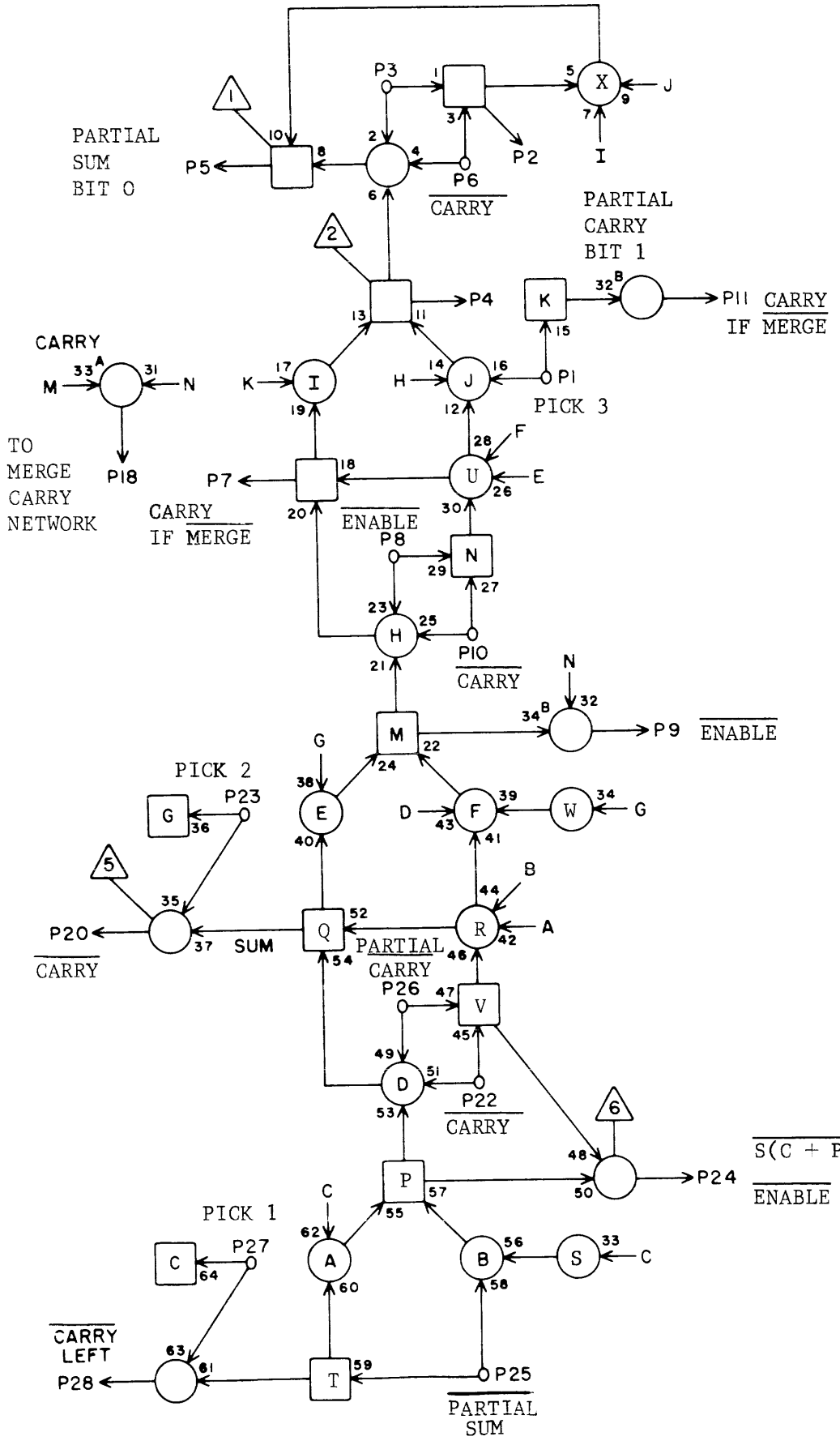
```
  0 0 0 0 0 1 0 1 0 0 1 1 1 0 0 1 0 1 1 1 0 1 1 1      1st pick
+ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0      partial sum
  0 0 0 0 0 1 0 1 0 0 1 1 1 0 0 1 0 1 1 1 0 1 1 1      sum

  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0      carrys
```

Figure 7.5-14

Notice that if either of the operands was a "1" (but not both) the answer for that bit position will be a 1, but if both operands in any single bit position were ones, the answer in that bit position must be zero and a carry to the next higher bit position will be generated. The circuitry which performs this addition is located on the MC and MG modules in the multiply unit. Refer to the schematic of the MC module (Figure 7.5-15). Each MC contains the complete adder for one bit position. The results (for one bit position) of the three picks and the partial sum and partial carry from the previous iteration are some of the inputs. The other inputs are carries from the next lower order MC. The outputs are carries to the next higher order MC and at the very end - the partial sum and partial carry for this iteration. These inputs and outputs will be defined as the discussion progresses.

The inputs associated with Figure 7.5-14 are at pins 25 and 27. Pin 27 receives the true value of the first pick and pin 25 the complement of the partial sum. Pin 28 will be zero when a carry is generated from the addition of these two numbers and P will be a "one" when either the first pick or the partial sum (but not both) are ones. Thus P represents the sum in Figure 7.5-14 and pin 28 represents the complement of the carry for any one bit position. The four possible cases are checked to prove the logic.

Figure 7.5-15



MC

	JACK	PIN	LG
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			
26			
27			
28			

The first case is for both the first pick and the partial sum equal to zero. Then pin 27 will be "0" and pin 25 a "1". First pin 27 forces pin 28 to one and since pin 28 is the complement of a carry, no carry is generated. Pin 25 caused T to go to a zero which in turn forces A to a one. Pin 27 causes S to be zero which in turn forces B to a one. With both A and B equal to 1, P will be zero.

The second case is for the first pick equal to a one and the partial sum equal to zero. Then pin 27 will be one and so will pin 25. Pin 28 will be forced to one (no carry) by T and B will be a zero forcing P to a one. Thus a sum of one and no carry result.

The third case is for the first pick equal to zero and the partial sum equal to one. The pin 27 = "0" and pin 25 = "0". Pin 27 forces pin 28 to one and A will be zero forcing P to a one, so that again, no carry and a sum of one result.

Notice in Figure 7.5-14 that the carries have been shifted left one bit indicating that they will be added to the next higher bit position during the next level of addition.

The second level of addition will add together the results of the first level (both pseudo sum and carries), the partial carries from the previous iteration, and the result of the second pick. This appears to be four numbers, however the carry from the first level of addition and the partial carry for any one bit position are mutually exclusive (one or the other, but never both can represent a one in any one bit position during a given iteration). The reason for this is that a particular bit position cannot generate a partial sum and partial carry of one. Thus if the partial carry

into a bit position is a one the partial sum into the first level of addition at the next lower order bit position must be zero. It was seen in the discussion of the first level of addition that both inputs must be ones to generate a carry to the next stage. Thus it is impossible to have both a partial carry and a carry from the next lower order stage equal to a one at the same time. Because of this these two inputs can be "ORed" and an addition of only three inputs for the second level of addition may be performed.

In Figure 7.5-16 the example started in Figure 7.5-14 is continued.

```
        0 0 0 0 0 1 0 1 0 0 1 1 1 0 0 1 0 1 1 1 0 1 1 1   Sum from previous
                                                          level of addition.
This  { 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   Carries from 1st
"or"                                                      level of addition.
this  { 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   Partial carries from
                                                          previous iteration.
+ 0 0 0 0 0 1 0 1 0 0 1 1 1 0 0 1 0 1 1 1 0 1 1 1 0       Result of 2nd pick
                                                          (2^2 & 2^3).
  0 0 0 0 0 1 0 1 1 0 0 1 1 1 1 0 0 1 0 1 1 0 0 1 1 1 1
                                                          Pseudo sum carries.
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 0 0 1 1 0 0 0 0
```

Figure 7.5-16

Notice that the second pick is two times the multiplicand and that it has been shifted left by two bits to indicate the bit position of the multiplier just as was done in Figure 7.5-13. Refer now to Figure 7.5-15. The pseudo sum enters this level of the adder in two places, P and A or B feeding R. The carry from the first level of addition (next lower order bit) enters at pin 22, as the complement of the true value. This is the carry generated at pin 28. The complement of the partial carry

enters at pin 26 and the true value of the second pick (2^2 & 2^3) enters at pin 23. Eight possible conditions must be considered since there are three inputs. In tabular form they are:

pseudo sum from previous level of addition	1 1 1 0 1 0 0 0	(A)
carry from previous stage or partial carry	1 1 0 1 0 1 0 0	(B)
second pick	1 0 1 1 0 0 1 0	(C)
pseudo sum from this level	1 0 0 0 1 1 1 0	
carry from this level	1 1 1 1 0 0 0 0	

Figure 7.5-17

The different inputs are denoted by A, B, and C as shown in Figure 7.5-17. Notice that a carry greater than one can never be generated from this level of addition. The carry can be generated at either pin 24 or pin 20 and thus both can never represent a carry of 1. A carry will be generated at pin 24 only when both A and B represent ones and a carry will be generated at pin 20 only when the sum of A and B is one and C is also a one. Then in Figure 7.5-17, the first two cases will generate a carry at pin 24 and the 3rd and 4th cases will generate a carry at pin 20. Finally, the pseudo sum is represented by the output of M and is sent to the next level of addition by M and by F and E feeding U. The mechanics of generating the pseudo sums and carries are left to the student except for the following example. In this case all three inputs will represent 1's. Then P must be a 1 and either A or B a zero. Either pin 22 or 26 must be zero, and pin 23 must be a 1.

Pin 22 or 26 will drive V and D to a 1 and V together with P will cause pin 24 to be a zero which represents a carry to the next higher order bit position. Since A or B is a zero, R will be a one and R and D cause Q to be zero. Q forces pin 20 to a 1 and thus there is no carry from pin 20. Q also forces E to a one. However, pin 23 was a 1 which causes W to be a one. D and R were also ones so F will be zero, and force U to a 1. Thus we have a pseudo sum of 1 at M and a carry of 1 at pin 24. This completes this level of addition.

The third level of addition once again has three inputs. They are the pseudo sum from the previous level of addition, the carry generated during the previous level of addition at the next lower order bit position and the result of the third pick. A carry generated at pin 24 for the next lower order MC feeds pin 8 and the carry generated at pin 20 feeds pin 10. It was seen that only one of these two pins can have a carry at any one time. Therefore the two carrys may be treated as one. Note, from the example that the third pick (2^4 & 2^5) is zero. This is added to the previous results as shown in Figure 7.5-18.

0 0 0 0 0 1 0 1 1 0 0 1 1 1 1 0 0 1 0 1 1 0 0 1 1 1 1	pseudo sum from 2nd level of adder
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 0 0 1 1 0 0 0 0	Carries from 2nd level of addition
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	Result of 3rd pick
0 0 0 0 0 0 0 1 0 1 1 1 0 1 1 1 0 0 0 0 0 1 0 1 0 1 1 1 1	pseudo sum
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0	

Figure 7.5-18

Notice that the result of the third pick has been shifted left two additional bits. Back in Figure 7.5-15 the result of the third pick enters the module at pin 1. The pseudo sum enters at M and F and E feeding V, and the carries at pins 8 and 10. The operation of this level of the adder is basically the same as the second level. The carry generated at pin 9 corresponds to the one generated at pin 24 and it feeds pin 3 of the next higher order MC. The carry corresponding to the one generated at pin 20 is actually generated on the MM modules as shown in Figure 7.5-19. It is logically the same as the carry generated at pin 20 and it feeds pin 6 of the next higher order MC. The reason for generating this carry on the MM module is for use during merge.

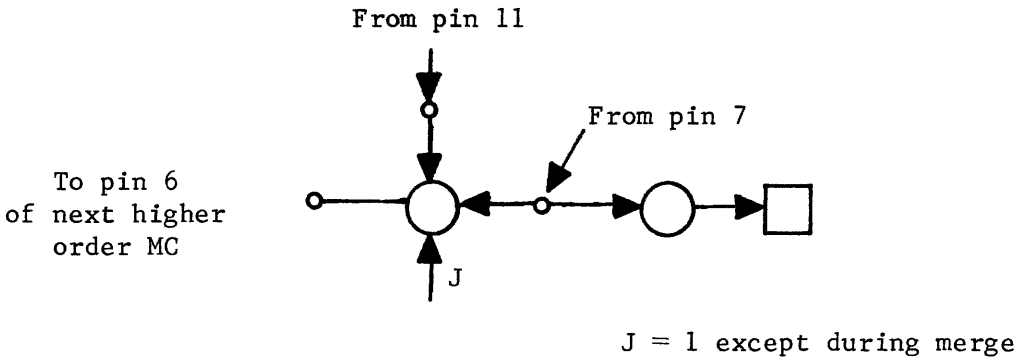


Figure 7.5-19

The pseudo sum is sent to the last level of addition by Test Point 2 and by I and J feeding X.

In the last level of addition, once again, there are just 2 numbers to add (the pseudo sum and the carries from the previous level of addition) just as in the first level of addition. It was seen that in the first level that both a pseudo sum of 1 and a carry of 1 at any one MC could not be generated because two inputs exist. The same is true here. The outputs of this final level of addition are called the partial sum and

the partial carry instead of the pseudo sum and carry, which refer to the outputs of the adder levels. The example in Figure 7.5-20 illustrates that a partial carry is generated in two bit positions and that the sum of the partial sums and partial carries actually does equal the result obtained in Figure 7.5-12 after the first iteration. Remember that the sum of the partial sums and carries is not actually taken, but that they are sent back to the MC's as two separate numbers for the next iteration.

```
0 0 0 0 0 0 0 1 0 1 1 1 0 1 1 1 0 0 0 0 0 1 0 1 0 1 1 1 1   pseudo sum.
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0     carries.
---------------------------------------------------------
0 0 0 0 0 0 0 1 0 1 1 1 0 1 1 0 0 0 1 0 0 0 0 1 0 1 1 1 1   partial sum.
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0     partial carry.
---------------------------------------------------------
0 0 0 0 0 0 0 1 0 1 1 1 1 0 0 0 0 0 1 0 1 0 0 1 0 1 1 1 1   sum of p.s.
                                                            and p.c.
0     0     1     3     6     0     2     4     5     7
```

Figure 7.5-20

On the MC module the partial sum is the true value of pin 5 and the partial carry is the "AND" of pins 2 and 4. The only condition that will generate a partial carry is when both the pseudo sum is 1 (TP = 1 and I or J = 0) and the carry is a one (pins 3 or 6 = 0).

The results of this iteration is sent to the F/Fs on the NJ modules to be stored for use during the next iteration. It was seen in the discussion of Figure 7.5-12 that the lowest 6 bits of the answer generated at the end of an iteration is the complete answer. The same is true here except that the answer is in two parts, the partial sum and partial carry. There is no need to put these six bits back into the adder for the next iteration. Instead, the partial sums and partial carries for these 6 bits will be added to obtain the actual sum. They are stored while the rest of the

multiply sequence finishes. There is a possibility that a carry can be generated from the upper bit position by the addition of these 6 bits of partial sum and partial carry. This will be discussed later. The remaining three iterations of the example follow.

```
                  0 0 0 0 0 0 0 1 0 1 1 1 0 1 1 0 0 0 1 0 0 0 0    1st pick (26 & 27)
                0 0 0 0 0 1 0 1 0 0 1 1 1 0 0 1 0 1 1 1 0 1 1 1    partial sum
                0 0 0 0 0 1 0 1 1 0 0 0 0 0 1 0 0 1 1 0 0 1 1 1    pseudo sums
              0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 1 0 0 0 1 0 0 0 0    carries
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0    partial carries
            0 0 0 0 0 1 0 1 0 0 1 1 1 0 0 1 0 1 1 1 0 1 1 1 0      2nd pick (2^8 & 2^9)
            0 0 0 0 0 1 0 1 1 0 0 0 0 1 1 1 0 1 0 1 1 1 1 1 0 1 1    pseudo sum
          0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 1 1 0 0 1 0 0 1 0 0      carries
          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0          3rd pick (2^10 & 2^11)
            0 0 0 0 0 0 1 0 1 1 1 1 1 1 0 1 1 1 1 0 0 1 0 1 1 0 0 1 1    pseudo sums
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0      carries
          0 0 0 0 0 0 0 1 0 1 1 1 1 1 1 0 1 1 1 0 0 0 0 0 1 0 0 0 1 1    partial sums
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0      partial carries
```

```
The partial sum is:      0 0 1 3 7 3 4 0 4 3
The partial carry is:    0 0 0 0 0 0 4 4 4 0
                         0 0 1 3 7 4 0 5 0 3
The lowest six bits from lst iteration          5 7
                         0 0 1 3 7 4 0 5 0 3 5 7
```

This answer check with the result if Figure 7.5-12 after the second iteration.

Notice that the sum of the lowest 6 bits of partial sum and carries does produce a carry.

```
  1 0 0 0 1 1
  1 0 0 0 0
 ↙0 0 0 0 1 1
1
```

This carry will be used at the end of the next iteration.

Figure 7.5-21, Second Iteration

```
            0 0 0 0 0 1 0 1 0 0 1 1 1 0 0 1 0 1 1 1 0 1 1 1    1st pick (2^12 & 2^13)
              0 0 0 0 0 0 0 1 0 1 1 1 1 1 0 1 1 1 0 0 0 0 0    partial sum
          -------------------------------------------------
            0 0 0 0 0 1 0 1 1 0 0 0 0 1 1 1 1 0 0 1 0 1 1 1    pseudo sum
          0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 1 1 0 0 0 0 0      carries
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0    partial carries
      0 0 0 0 0 1 0 1 0 0 1 1 1 0 0 1 0 1 1 1 0 1 1 1 0        2nd pick (2^14 & 2^15)
    -------------------------------------------------------
      0 0 0 0 0 1 0 1 1 0 0 0 0 1 1 1 1 0 0 1 1 0 0 1 0 1 1    pseudo sum
    0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 1 1 1 0 1 1 0 1 0 0      carries
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0            3rd pick (2^16 & 2^17)
   --------------------------------------------------------
    0 0 0 0 0 0 1 0 1 1 1 1 1 0 1 1 1 0 1 1 1 0 1 0 0 0 1 1    pseudo sum
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 1 0 0 0      carries
 ----------------------------------------------------------
  0 0 0 0 0 0 0 1 0 1 1 1 1 1 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1    partial sum
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1    partial carries
                                                          ^
                                                          |
```

<u>NOTE</u> This bit is the carry left over from the previous iteration after adding together the lowest 6 bits of partial sums & carries.

```
                         The partial sum is:     0 0 1 3 7 3 1 4 6 3
                         The partial carry is:   0 0 0 0 0 1 2 4 0 1
                                                 -------------------
                                                 0 0 1 3 7 4 2 0 6 4
                 The lowest 12 bits from the
                 1st & 2nd iterations are:                           0 3 5 7
                                                 ---------------------------
                                                 0 0 1 3 7 4 2 0 6 4 0 3 5 7
```

This answer again agrees with the answer obtained at the end of the third iteration if Figure 7.5-12.

The carry, described in the note, was placed in the NJ modules which hold the partial sums and carries at the same time as the other bits of the partial sums and carries for the third iteration were placed there. This is how a carry from the lowest six bits of answer at the end of an iteration is propagated. It will now be added to the lowest six bits of the partial sum along with the other five lowest bits of partial carry to obtain the lowest six bits of answer for this iteration.

Figure 7.5-22, Third Iteration

```
            0 0 0 0 0 1 0 1 0 0 1 1 1 0 0 1 0 1 1 1 0 1 1 1   1st pick
            0 0 0 0 0 0 0 0 1 0 1 1 1 1 1 0 1 1 0 0 1 1 0 0   partial sum
          -------------------------------------------------
            0 0 0 0 0 1 0 1 1 0 0 0 0 1 1 1 1 0 1 1 1 0 1 1   pseudo sum
          0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 1 0 0 0 1 0 0     carries
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0   partial carries
      0 0 0 0 0 1 0 1 0 0 1 1 1 0 0 1 0 1 1 1 0 1 1 1 0       2nd pick
    -------------------------------------------------------
      0 0 0 0 0 1 0 1 1 0 0 0 0 1 1 1 1 0 0 1 1 0 0 1 1 1 1   pseudo sum
    0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 1 1 1 0 1 1 1 0 0 0     carries
    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0           3rd pick
   --------------------------------------------------------
    0 0 0 0 0 0 1 0 1 1 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 1 1 1   partial sum
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0     carries
 ----------------------------------------------------------
  0 0 0 0 0 0 0 1 0 1 1 1 1 1 0 1 1 0 0 1 1 0 0 1 1 1 1 1 1   partial sum
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0     partial carries
```

```
The partial sum is:         0 0 1 3 7 3 1 4 7 7
The partial carries are:    0 0 0 0 0 1 0 4 0 0
                            -------------------
The sum of the two is:      0 0 1 3 7 4 2 0 7 7
The lowest 18 bits from
the 1st three iterations are:                   6 4 0 3 5 7
                            -----------------------------------
                            0 0 1 3 7 4 2 0 7 7 6 4 0 3 5 7
```

This is the answer obtained if Figure 7.5-12 at the end of the fourth iteration.

Figure 7.5-23, Fourth (& last) iteration

In performing the four iterations, a 24 bit multiplicand was assumed. However, the actual multiplicand in the 6600 is 48 bits. If the extention of zero's and additional 24 bits left is assumed, the example will hold true and the results obtained will be the lower product after 4 iterations. If the upper 24 bits of the multiplier were the same as the lower 24 bits, the upper product would be identical with the lower product. This case is assumed, and the original problem is actually:

```
                                 0 0 0 0 0 0 0 0 0 0 1 2 3 4 5 6 7
                               x 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                               -----------------------------------
0 0 0 0 0 0 0 0 0 0 1 3 7 4 2 0 7 7 7 7 7 7 7 7 7 7 7 6 4 0 3 5 7
```

Then to obtain the full answer, the upper and lower products must be added together as follows:

```
                    0 0 0 0 0 0 0 0 0 0 0 1 3 7 3 1 4 7 7 6 4 0 3 5 7  Lower p.s.
                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 4 0 0              Lower p.c.
  0 0 0 0 0 0 0 0 1 3 7 3 1 4 7 7 6 4 0 3 5 7                          Upper p.s.
  0 0 0 0 0 0 0 0 0 0 0 1 0 4 0 0                                      Upper p.c.
-------------------------------------------------------------------------------
0 0 0 0 0 0 0 0 0 0 1 3 7 4 2 0 7 7 7 7 7 7 7 7 7 7 6 4 0 3 5 7  Final product
```

Notice that the lowest 18 bits of the answer are complete and that the next 6 bits consist of the partial sum and partial carry for the lower product only. It is not necessary that these bits be placed in the adder for merge. The only exception is if the upper 6 bits of the 24 bits (77-00 in our example) were to produce a carry, that carry must be placed into the merge circuitry. These six bits are added together in the same circuitry that was used for the other 3 groups of 6 bits at the end of each iteration. If a carry is produced it is introduced into the merge circuitry (at about half way through merge).

The merge addition uses the Lower 3-Level Adder (MC modules) and a special carry network. Notice that the lowest 18 bits of the upper product are the complete answer (for the upper product) when they enter the merge circuitry. In other words, there is no partial carry for these bits.

The final operation involved in multiply is merge. While the four iterations of the lower product were being performed, the upper product was also being produced. It will produce an answer in a similar manner to that of the lower product circuitry. The only major difference is that the lowest 6 bits of the answer are left in the form of partial sum and

partial carries when it is fed into the merge circuitry. Thus at the end of the fourth iteration the lower product in our example will be in the form:

```
0 0 1 3 7 3 1 4 7 7 6 4 0 3 5 7    (PS)
0 0 0 0 0 1 0 4                    (PC)
```

and the upper product has the form:

```
0 0 1 3 7 3 1 4 7 7 6 4 0 3 5 7    (PS)
0 0 0 0 0 1 0 4 0 0                (PC)
```

Actually, the lowest six bits of partial sum and carries for the lower product have not been added together at the time merge starts. However they are added together in the same circuitry (Lower Six Bit Adder) that was used at the end of the first three iterations. If a carry is produced from this addition it is fed into the merge circuitry as soon as it is produced. Thus the merge circuitry performs the following full add.

```
                  0 0 1 3 7 3 1 4
                  0 0 0 0 0 1 0 4
0 0 1 3 7 3 1 4 7 7 6 4 0 3 5 7
0 0 0 0 0 1 0 4 0 0
-------------------------------
0 0 1 3 7 4 2 0 7 7 7 7 7 7 7 7
```

Appending the lower 24 bits produces the correct answer:

```
0 0 0 0 0 0 0 0 0 0 1 3 7 4 2 0 7 7 7 7 7 7 7 7 7 7 6 4 0 3 5 7
```

The inputs to the MC's and G modules during merge are as follows:

Pin 27 - The upper product partial sum (or in the case of the lowest 18 bits of upper product - the full sum).

Pin 25 - The lower product partial sum.

Pin 26 - The lower product partial carries.

Pin 23 - The upper product partial carries. (For the lowest 18 bits this input is zero since pin 27 has the full sum.)

Pin 1 - Carry generated in the carry network.

The other inputs are carries generated within the adder and passed up to the next higher bit position just as in the four iterations.

The outputs are as follows:

Pin 18 - (Pin 9 is the same output) goes to carry network.

Pin 11 - Not used.

Pin 7 - To carry network.

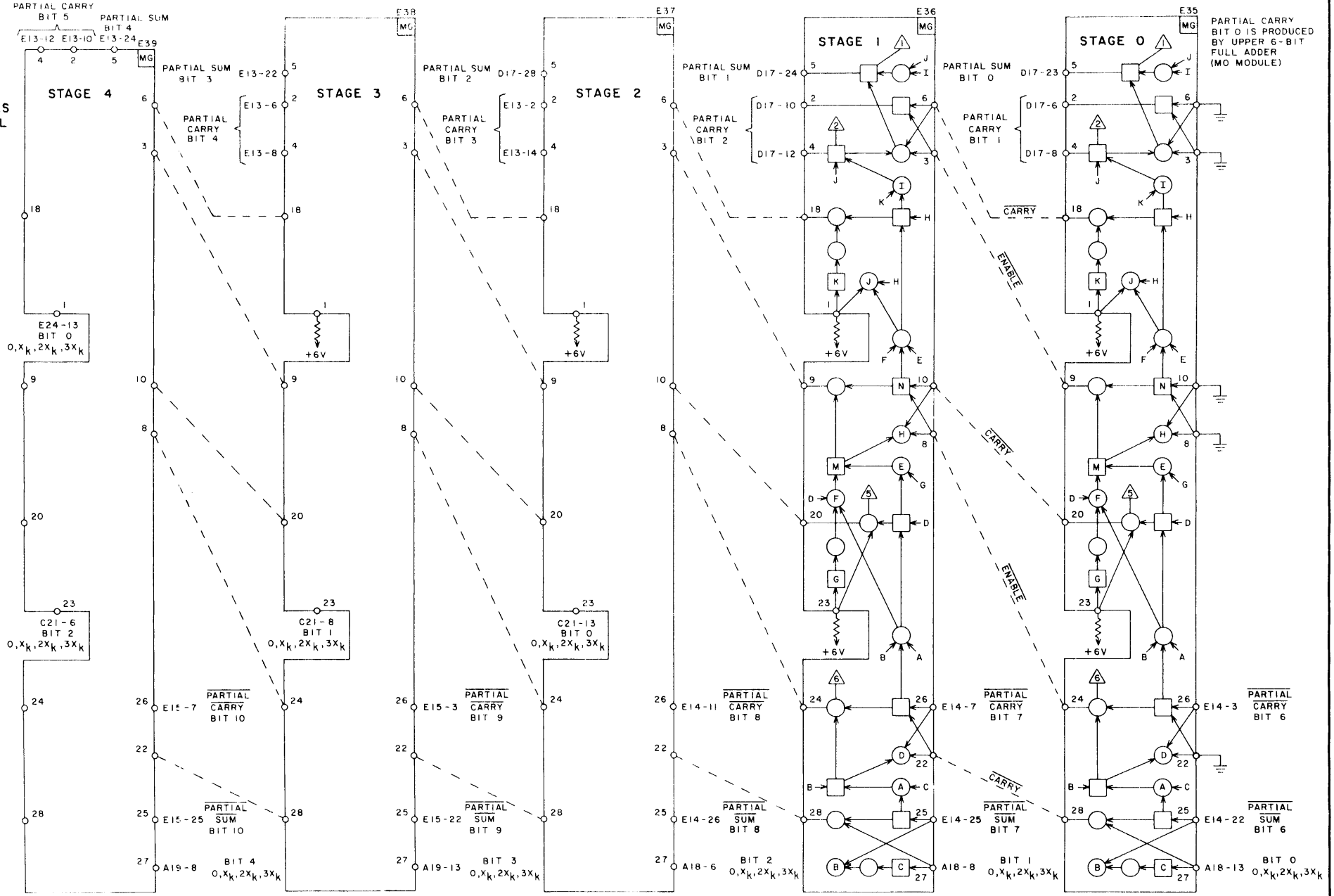
Pin 5 - Final output of adder.

The three level adder is capable of producing only 57 bits (24-80) of the final product during Merge. The 4 lower iterations produced the lower 24 bits of the final product. This leaves 15 bits (81-95) which are produced by a special 15 bit adder (discussed in Section 7.5.10).

Figures 7.5-24 and 7.5-25 are representative drawings of the Upper and Lower Three Level Adders, respectively. Note that the upper adder is composed of MG modules, and the lower, which is used during Merge, of MC modules. The MC's have additional carry outputs to the Merge Carry Network (from the second level) and inputs from the same network (into

level three).* Since the operation of all 57 stages is identical these diagrams should be sufficient to understand the adder operation. If more specific information on the other stages is desired, reference should be made to the Chassis #6 wire tabs.

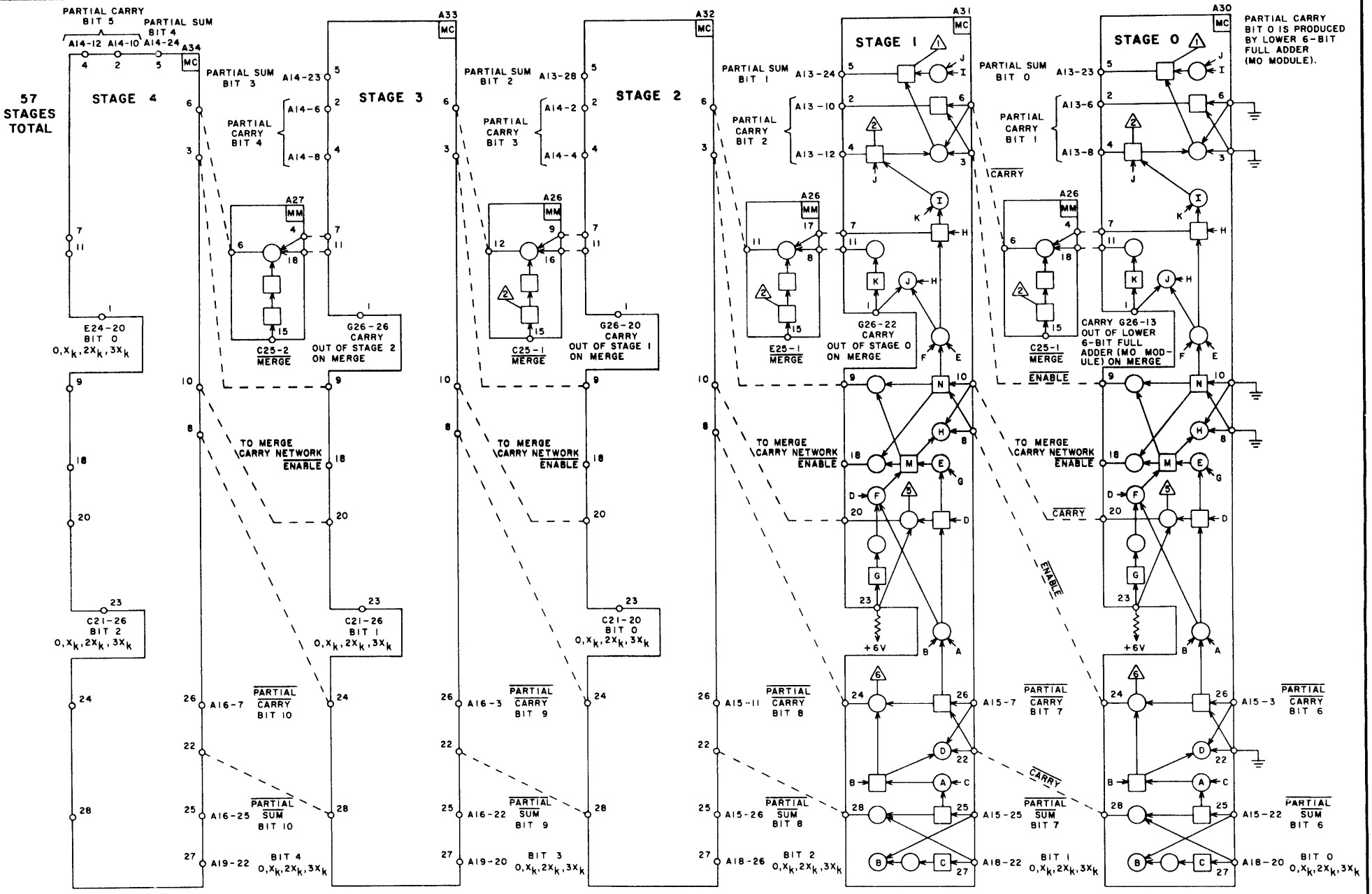
* All inputs to the Lower Three Level Adder are summarized according to pin number and bit position in Figure 7.5-29.



NOTES:

1. ALL LOCATIONS ON CHASSIS 6.
2. MULTIPLY 1 SHOWN, 2 IS SIMILAR.

Figure 7.5-24



NOTES:

1. ALL LOCATIONS ON CHASSIS 6.
2. MULTIPLY 1 SHOWN, 2 IS SIMILAR.

Figure 7.5-25

7.5.7 SIX BIT ADDERS

The purpose of the Six Bit Adders is to fully add the lower six bits of the upper and lower Partial Sum and Partial Carry Registers after each iteration of the multiply step. Since the adders are identical in operation, only the Upper Adder (Figure 7.5-26) is discussed. The logic for the Lower Adder is shown in the C. E. Diagrams, Sheet 153.

Since only six bits are being added and a full minor cycle is allowed for this addition (during which the next iteration is taking place in the three-level adders), the time required to propagate a carry is not critical. The propagation is therefore, serial, as opposed to the parallel carry summation used in 6600 adders having a greater modulus.

In Figure 7.5-26 are shown the Partial Sum and Carry registers which feed the adder logic (MO modules). The bits are fed via MM modules which provide the true and false values of PS and PC for the MOs.

The true value of the sum is taken at output pins 9, 1, 28, 21, 20 and 12 for bits 0, 1, 2, 3, 4 and 5, respectively. Boolean formulas are used to express the conditions for a sum = 1 in a given bit position. For simplification, the following abbreviations are used in the formulas:

PSX $\Rightarrow$ Partial Sum in the feeder for bit 2^X.
PCX $\Rightarrow$ Partial Carry in the feeder for bit 2^X.
CX $\Rightarrow$ Carry <u>from</u> stage 2^X <u>into</u> $2^X + 1$.

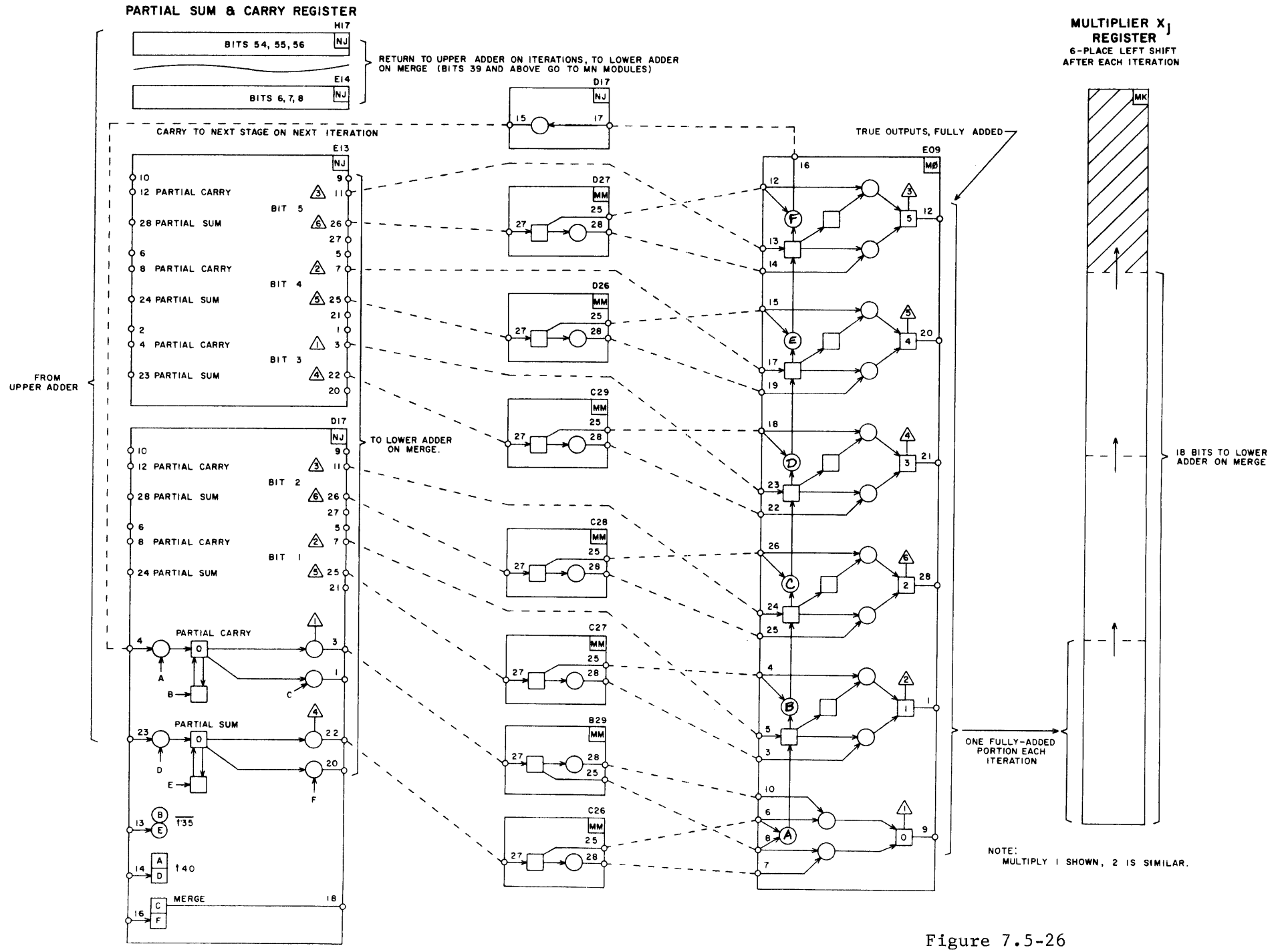

PARTIAL SUM & CARRY REGISTER
HI7
BITS 54, 55, 56
NJ
E14
BITS 6, 7, 8
NJ
RETURN TO UPPER ADDER ON ITERATIONS, TO LOWER ADDER ON MERGE (BITS 39 AND ABOVE GO TO MN MODULES)
MULTIPLIER X_J REGISTER
6-PLACE LEFT SHIFT AFTER EACH ITERATION
MK
CARRY TO NEXT STAGE ON NEXT ITERATION
TRUE OUTPUTS, FULLY ADDED
FROM UPPER ADDER
E13
NJ
10
12 PARTIAL CARRY
28 PARTIAL SUM
6
8 PARTIAL CARRY
24 PARTIAL SUM
2
4 PARTIAL CARRY
23 PARTIAL SUM
BIT 5
BIT 4
BIT 3
D17
NJ
BIT 2
BIT 1
TO LOWER ADDER ON MERGE.
PARTIAL CARRY
PARTIAL SUM
MERGE
D17
NJ
D27
MM
D26
MM
C29
MM
C28
MM
C27
MM
B29
MM
C26
MM
EO9
MØ
ONE FULLY-ADDED PORTION EACH ITERATION
18 BITS TO LOWER ADDER ON MERGE
NOTE:
MULTIPLY 1 SHOWN, 2 IS SIMILAR.

Figure 7.5-26

The inverters labeled A, B, C, D, E and F, when a "zero", indicate a carry out of that stage. In other words,

$A \Rightarrow \overline{C0}$ $\quad$ $D \Rightarrow \overline{C3}$

$B \Rightarrow \overline{C1}$ $\quad$ $E \Rightarrow \overline{C4}$

$C \Rightarrow \overline{C2}$ $\quad$ $F \Rightarrow \overline{C5}$

In general, a carry out of a stage occurs if that stage generates a carry (i.e. PSX · PCS $\Rightarrow$ CX) or if there is a carry into that stage and that stage has a one in its Partial Sum flip-flop (i.e. PSX · CX-1 $\Rightarrow$ CX). The case where a carry into a stage occurs and that stage has its Partial Carry flip-flop set is not possible because in order to generate a carry, a stage must have a one in its PS flip-flop. This prohibits the next significant stage from having its PC flip-flop set since a PSX and PCX + 1 from the Three Level Adder is not possible. (This was proven in the discussion of the Three Level Adder, Section 7.5.6). Hence the general case formula for a carry out of Stage X is:

$$PSX(PCX + CX\text{-}1)$$

The following are formulas which express the conditions that result in a Sum = 1 for each stage. The formulas should be proven by the reader with application of Boolean Algebra to the logic circuits of the MO modules.

Bit	Pin	TP	Formula For Sum = 1
2^0	9	1	$PS0 \cdot \overline{PC0} + \overline{PS0} \cdot PC0$
2^1	1	2	$PS1 \cdot \overline{PC1} \cdot \overline{C0} + \overline{PS1}(PC1 + C0)$
2^2	28	6	$PS2 \cdot \overline{PC2} \cdot \overline{C1} + \overline{PS2}(PC2 + C1)$
2^3	21	4	$PS3 \cdot \overline{PC3} \cdot \overline{C2} + \overline{PS3}(PC3 + C2)$
2^4	20	5	$PS4 \cdot \overline{PC4} \cdot \overline{C3} + \overline{PS4}(PC4 + C3)$
2^5	12	3	$PS5 \cdot \overline{PC5} \cdot \overline{C4} + \overline{PS5}(PC5 + C4)$

Note the output at E09, pin 16 (term F $\Rightarrow\overline{C5}$) which is fed back to D17 to the Partial Carry flip-flop for bit 2^0. This occurs so the possible carry from the 6-bit adder is added during the next interation of the Multiply step.

7.5.8 MERGE

At the completion of the fourth multiply iteration, the product is spread throughout several registers in different forms. (Refer to Figure 7.5-27.)

1) The MK Lower register holds the lower 24 bits (0-23) of the fully added product.

2) The MK Upper register holds the next 18 bits (24-41) of the fully added product (which may be modified by a carry from bit 23).

3) The Lower Partial Sum register holds 47 bits of unsummarized sums.

4) The Lower Partial Carry register holds 48 bits of unsummarized carries (including the possible carry from the fourth Lower Six Bit Add in 2^0).

5) The Upper Partial Sum register holds 48 bits of unsummarized sums.

6) The Upper Partial Carry register holds 48 bits of unsummarized carries.

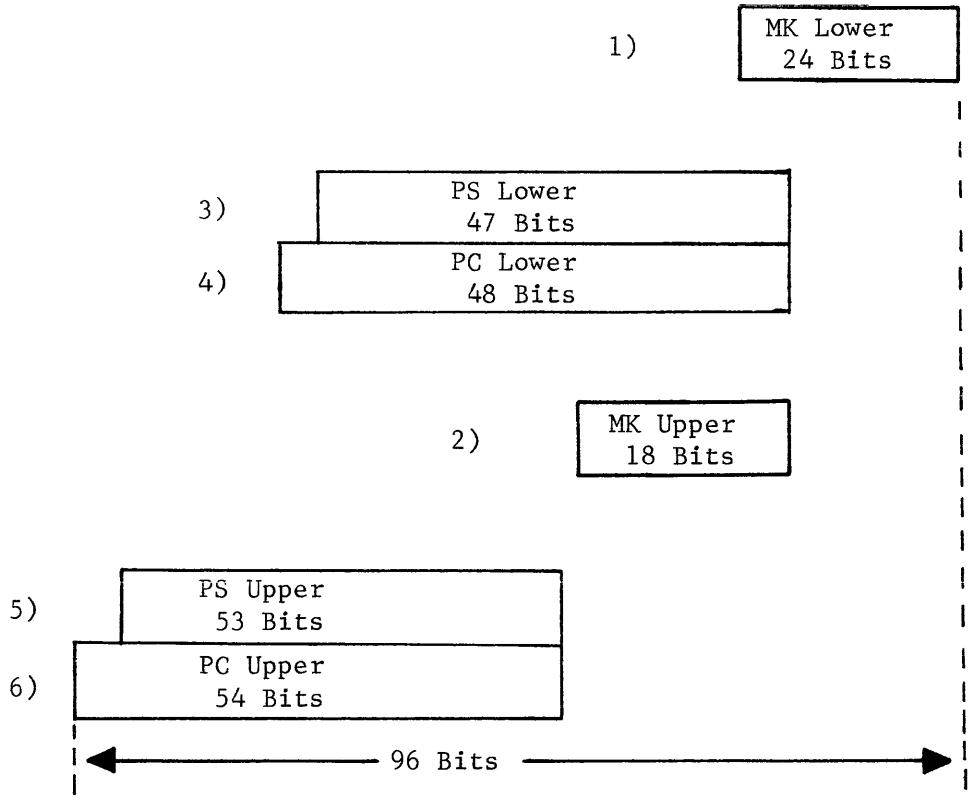
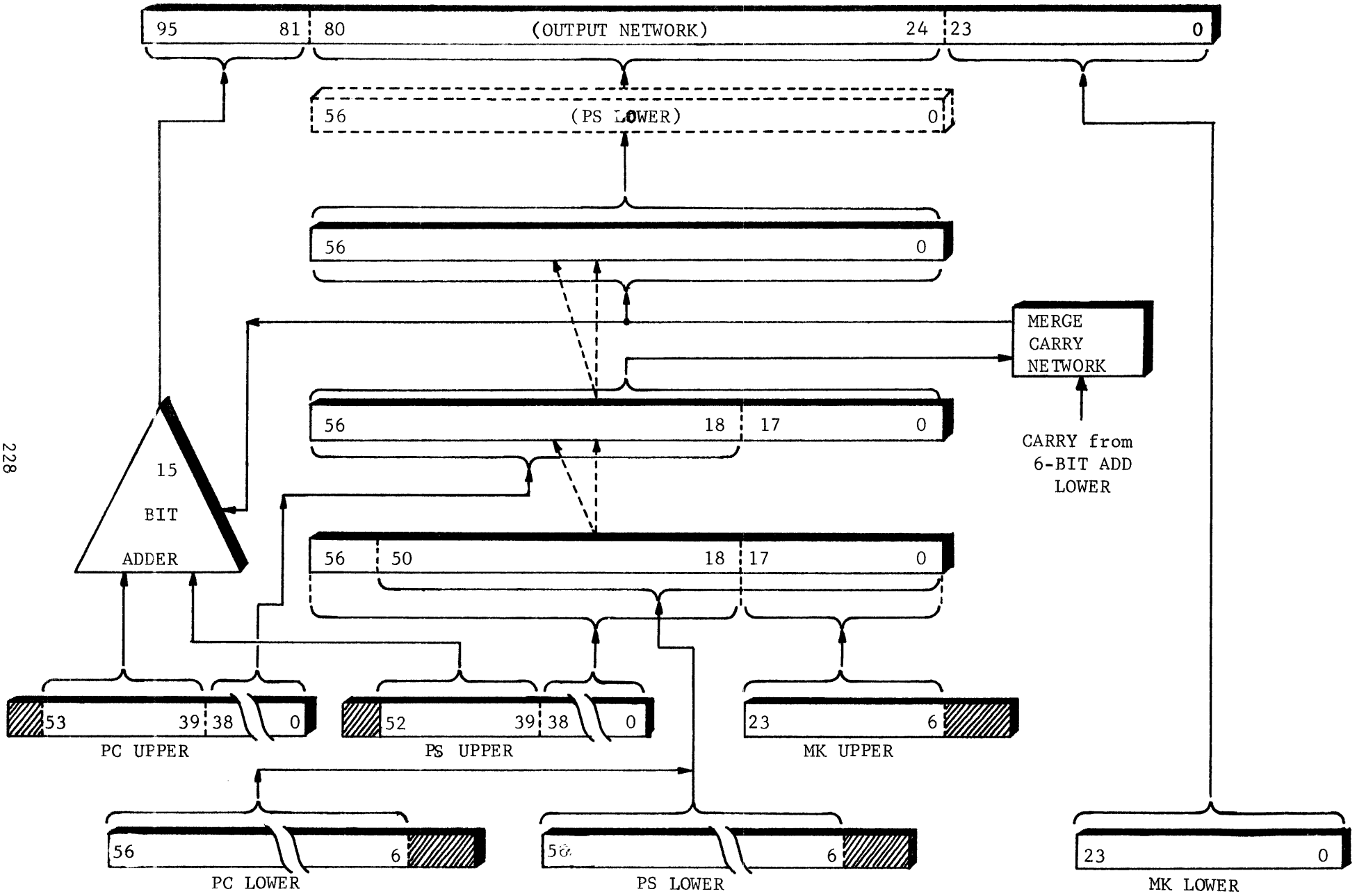


Figure 7.5-27

Figure 7.5-28 - MERGE BLOCK DIAGRAM



Combining these values to form the final 96-bit product is the function of the Merge operation.

Since the content of MK Lower is a fully added value, it need not be summarized and can be fed directly to the output network. (Refer to Figure 7.5-28.)

The 57 stages of the Lower Three Level Adder are used to summarize the following values:

1) The 18-bits of MK Upper (2^{6}-2^{23}).
2) The 51-bits of PS Lower (2^{6}-2^{56}).
3) The 51-bits of PC Lower (2^{6}-2^{56}).
4) The lower 39-bits of PS Upper (2^{0}-2^{38}).
5) The lower 39-bits of PC Upper (2^{0}-2^{38}).

The remaining bits of PS Upper (2^{39}-2^{52}) and PC Upper (2^{39}-2^{53}) are added in a special 15-bit full adder which is discussed in Section 7.5.9.

Since the Three Level Adder logic works essentially the same during Merge as during normal iterations (with the exception of the Full Add performed in level three) the adder itself is not discussed at this point. (It is discussed, along with the Merge Carry Network, in Section 7.5.6.) The fact that all unsummarized bits of the product mentioned above are fed into the adder during Merge is shown by Figure 7.5-29.

This figure indicates the values fed to the various input pins of the MC modules according to bit position. Since the information on the chart can be proven only by use of the Chassis #6 wire tabs further discussion of these inputs is not made, with the following

INPUTS TO LOWER THREE LEVEL ADDER

PIN	NORMAL ITERATIONS		MERGE	
	BITS	SOURCE	BITS	SOURCE
1	0-3 4-54 55-56	"Zeros" Pick 3 "Zeros"	0 1-56	"Zero" Carry from Merge Carry Network
3	0 1-56	$\overline{\text{\quad}}$ "One" $\overline{\text{Carry}}$ from previous stage (Level 2)	0 1-56	$\overline{\text{\quad}}$ "One" $\overline{\text{Carry}}$ from previous stage (Level 2)
6	0 1-56	$\overline{\text{\quad}}$ "One" $\overline{\text{Carry}}$ from previous stage (Level 2)	0-56	"Ones"
22	0 1-56	$\overline{\text{\quad}}$ "One" $\overline{\text{Carry}}$ from previous stage (Level 1)	0 1-56	$\overline{\text{\quad}}$ "One" $\overline{\text{Carry}}$ from previous stage (Level 1)
23	0-1 2-52 53-56	"Zeros" Pick 2 "Zeros"	0-17 18-52 53-56	"Zeros" Upper PC Register, bits 0-34 "Zeros"
25	0-50 51-56	$\overline{\text{Lower PS Register, bits 6-56}}$ "Ones"	0-50 51-56	Lower PS Register, $\overline{\text{bits 6-56}}$ Upper PS Register, $\overline{\text{bits 33-38}}$
26	0-50 51-56	$\overline{\text{Lower PC Register, bits 6-56}}$ "Ones"	0-50 51-56	Lower PC Register, $\overline{\text{bits 6-56}}$ "Ones"
27	0-50 51-56	Pick 1 "Zeros"	0-17 18-50 51-52 53-56	Upper MK Register, bits 6-23 Upper PS Register, bits 0-32 "Zeros" Upper PC Register, bits 35-38

Figure 7.5-29

exception. Those inputs which, during Merge, use the same input as the Pick 1, 2, and 3 inputs (i.e. pins 1, 23 and 27) during normal iterations are not entered directly, but are fed through the 1, 2 or 3Xk selection circuits. Examples of this are shown in Figure 7.5-11. The remaining merge inputs are fed directly from the source to the adder.

7.5.9 FIFTEEN BIT ADDER:

The Fifteen Bit Adder is used during the merge phase of multiply to add the upper fifteen bits (39-53) of the Upper Partial Sum and Partial Carry registers. (See Figure 7.5-28.)

This circuit is a true adder in that true values are held in the feeders and the true sum is seen at the output. Generates, Satisfies and Enables are therefore defined as follows:

Generate:	Satisfy:	Enables:
1	0	1 0
1	0	0 1

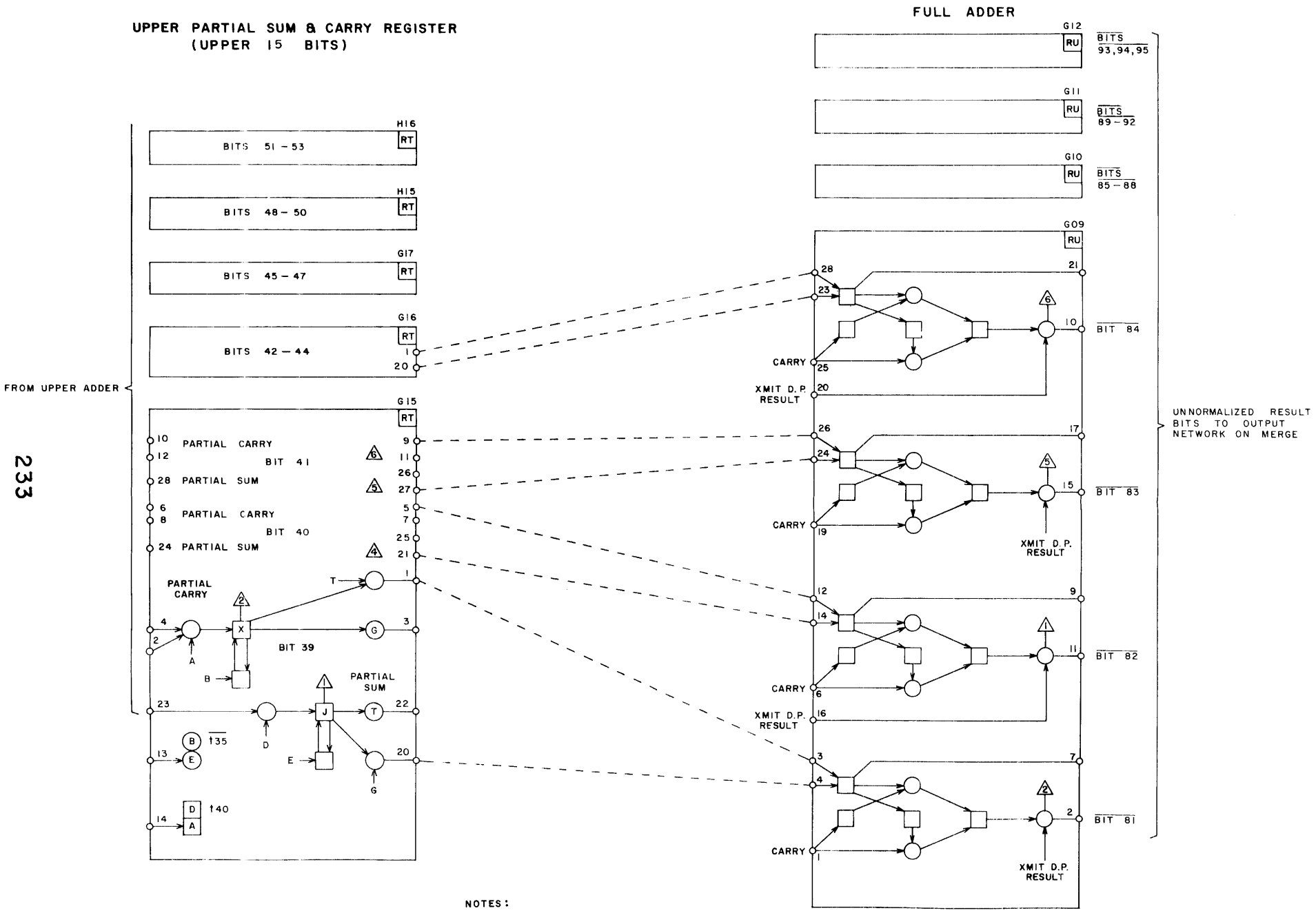
Figure 7.5-30.1 shows the feeder and final summation logic for the Fifteen Bit Adder. The carry logic is omitted, but uses the standard pass and carry check method of carry propagation.

The final summation logic (RU modules) generate "ones" and "zeros" in the final sum according to the following formulas:

$$\text{EQUIVALENCE} \cdot \text{CARRY} + \overline{\text{EQUIVALENCE}} \cdot \overline{\text{CARRY}} \Rightarrow 1$$

$$\text{EQUIVALENCE} \cdot \overline{\text{CARRY}} + \overline{\text{EQUIVALENCE}} \cdot \text{CARRY} \Rightarrow 0$$

The square fed by pins 3 and 5 on G09 (bit 2^{81}) translates as $\overline{\text{EQUIVALENCE}}$. Pin 1 is the carry input from bit 2^{80} and is generated by the Merge Carry Network (i.e. E29, pin 21). The true output is seen at the input to test point 2. At this point, translation yields the above formulas, so the output of the test points is the complement of the true result. This value (possibly left-shifted to normalize) is fed to the transmitters for the Multiply data trunk, bits 81-59, if Single Precision is selected.



NOTES:

1. LOCATIONS ON CHASSIS 6.
2. MULTIPLY 1 IS SHOWN, MULTIPLY 2 IS SIMILAR.

7.5-30.1

7.5.10 EXPONENT TIMING SEQUENCE

As with the coefficient logic, the exponent logic is duplicated for Multiply Units I and II. The registers which feed the Xj + Xk Adder are exceptions since they are common to the Multiply 1, Multiply 2 and Divide Units. (See Figure 7.5-30.2)

Since a general discussion of exponent manipulation appears in Section 7.5.1, the concepts are not reiterated at this point. This section, then, deals primarily with the logic analysis.

The timing chain for a Multiply Unit (Multiply 1 will be discussed) is composed of a chain of flip-flops which are set by various clock times as indicated on the Exponent Timing Chart (Figure 7.5-31). Since each flip-flop is clear/set with a standard timing pulse, each is set for approximately one minor cycle. (Refer to the C.E. Diagrams, Sheets 157 & 158 for the timing chain logic.)

In Figure 7.5-31, the references given in parenthesis following each term name refer to the output pins from the timeing sequence shown on Sheets 157 & 158 of the C. E. Diagrams. On the other hand, the pulses indicate the approximate time that the specified gate actually occurs. The times listed across the top of the chart are with reference to the Coefficient Timing Chain (Figure 7.5-7). The following is an explanation of each term shown on the timing chart:

1) Go, Multiply 1 Exponent - This term shown the time that the "Go Multiply" is received from Chassis #6. (See Figure 7.5-6.)

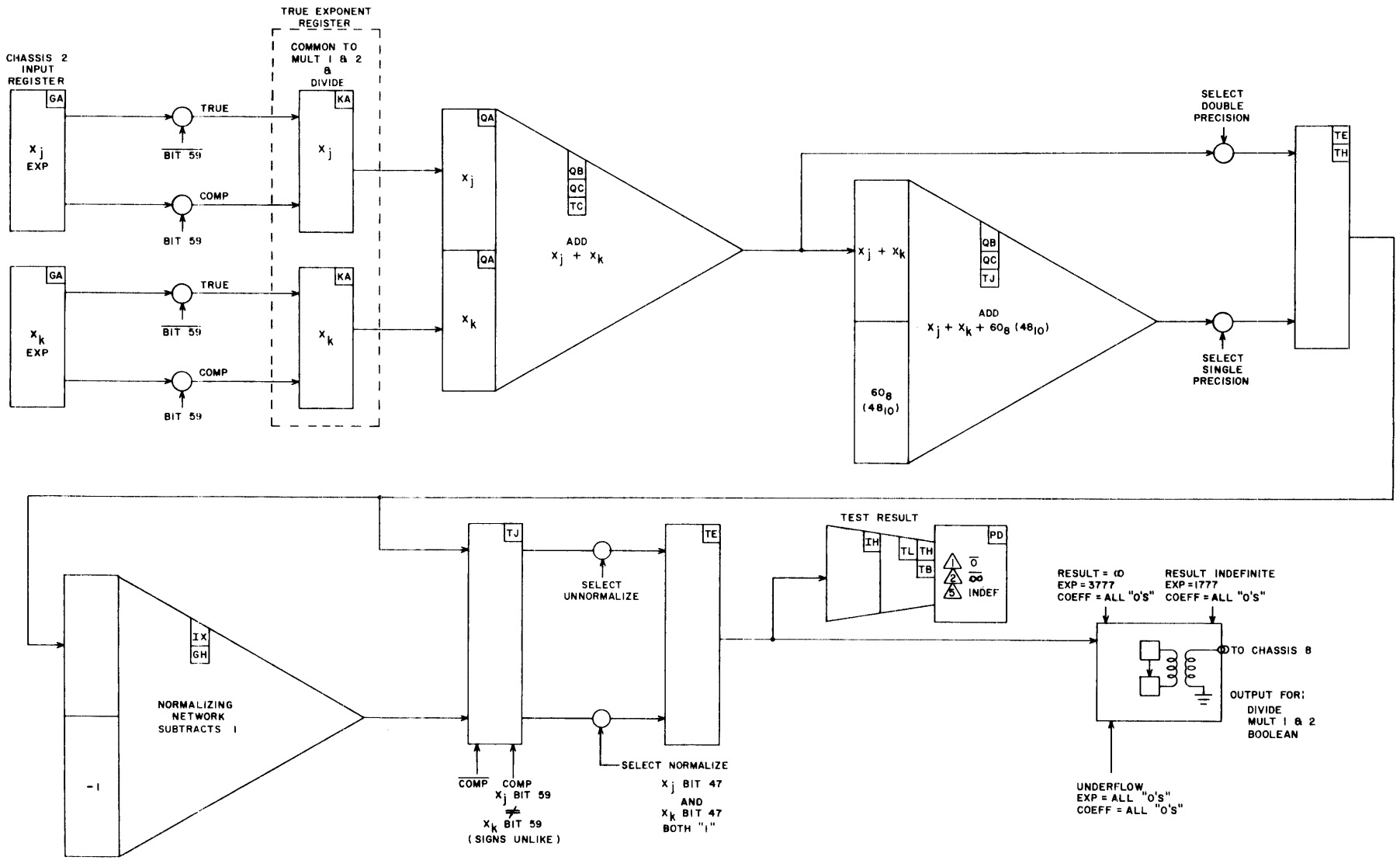


Figure 7.5-30.2

2) Advance Timing Sequence - This term shows the sequential setting of the five flip-flops in the exponent timing chain, as follows:

t375	-	D28, TP2
t450	-	D28, TP5
t500	-	J32, TP1
t575	-	J32, TP4
t650	-	E17, TP1

3) Test Xj & Xk Signs - This term checks the signs (2^{59}) of the Xj and Xk operands and enables the selection of the true (if $2^{59} = 0$) or false (if $2^{59} = 1$) value of the Input register. The IR is gated to the holding register at t375 by term #5. This is the first step in unpacking the exponents. (See Section 7.5.1 where exponent manipulation is discussed.)

4) Clear Normalize & Double Precision Flip-Flops - These flip-flops are cleared in preparation for the receipt of the "Normalize" or "D. P." signals from Chassis 6. (2Q11,TP2 & 1.)

5) Xj & Xk IR → Holding Registers - This gate enables the transfer of the Input Register (bits 48-59) to the holding register of Chassis 2. It is with this transfer that bits 2^{58} of Xj and Xk are extended to bits 2^{59}.

6) Test Non-Standard Operands - This term gates bits 48-59 of Xj and Xk into registers which feed the Error Test Circuitry. This logic is shown in the C. E. Diagrams, Sheets 93 and 94.

7) Select True Exponent at the Holding Register - This gate enables the selection of the true value from the holding register.

8) Select True Or Complement Result - This gate enables the setting of the TRUE or COMPLEMENT flip-flops on module H32, via pin 9 (C. E. Diagrams, Sheet 165). True is selected if the original signs are alike; Complement if unlike. The flip-flops will determine whether or not the packed final exponent is gated out in true or in complement form.

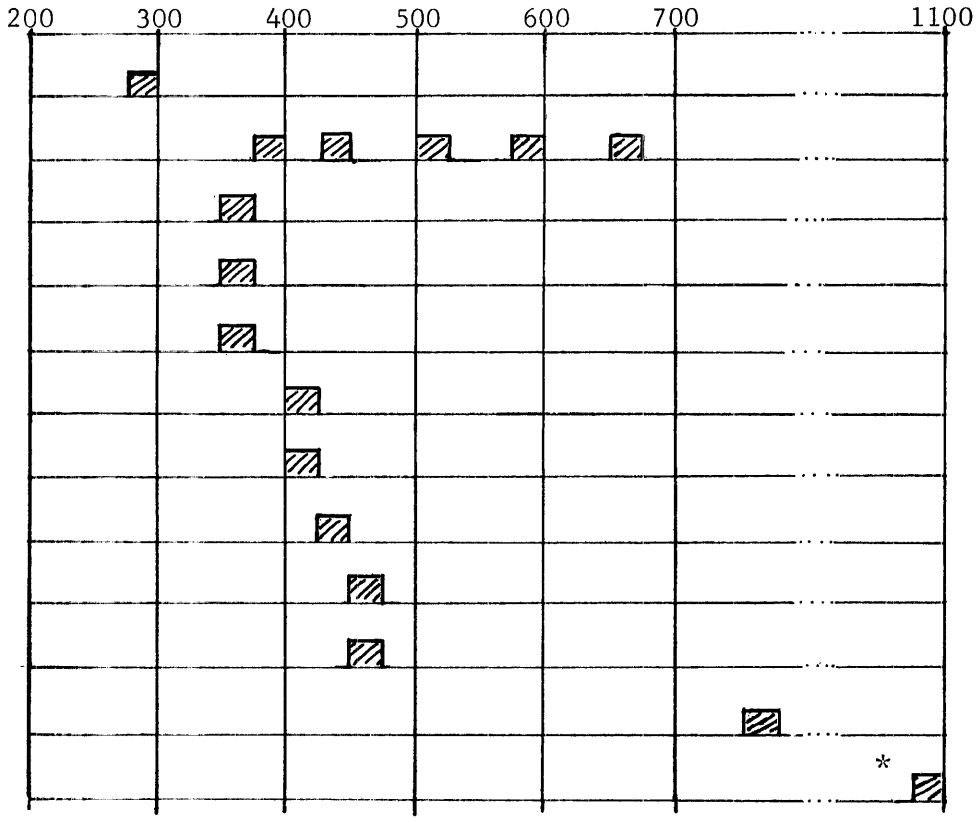
9) Holding Register Complement → Feeder - This gate transfers the complement of the Xj and Xk exponents (unpacked) to the adder feeder registers (QA modules).

10) Set Non-Standard Operand Flip-Flops - This gate sets the zero, infinite, and indefinite flip-flops on module E20. These determine whether or not to 1) send an Error indication to Chassis 5 and 2) force the Non-standard opcode bit configurations in bits 48-59 of the result (i.e. 0000, 1777, 3777 or 4000).

Figure 7.5-31

EXPONENT TIMING SEQUENCE - CHASSIS 2

1) Go, Multiply 1 Exponent (M11,TP2)
2) Advance Timing Sequence
3) Test Xj & Xk Signs (A36,14)
4) Clr. Norm. & D.P. f.f. (A36,9)
5) Xj & Xk IR → Holding Reg. (M11,21)
6) Test Non-Standard Operands (A36,13)
7) Select True Exp. @ Holding Reg. (D28,11)
8) Select True or Comp. Result (D28,27)
9) Holding Reg. Comp. → Feeder (I32,10)
10) Set Non-Standard Operand f.fs. (I32,10)
11) Mult. 1 Error → Chassis 6 (B35,5,7)
12) Transmit Result (RMs)



* Earliest possible time - no third order conflicts.

11) Multiply Error ⟶ Chassis 6 - This term enables the Transmitters (J03) which send the ERROR indication to Chassis 5.

12) Transmit Results - This indicates the earliest possible time (no third order conflicts) that the result exponent can be gated to Register Exit/Entry Control.

7.5.11 EXPONENT ADDERS

Three adders, shown in Figure 7.5-30.2 are required for the formation of the final exponent.

The first forms the algebraic sum of Xj and Xk. This value will be used as the final exponent if Double Precision is selected and left shifting to normalize is not performed.

The second forms the sum of adder #1 and 60_8 (48_{10}). This value will be used as the final exponent if Single Precision is selected and left shifting to normalize is not performed.

The third subtracts 1 from the result of the first or second adders (depending upon Single or Double Precision mode, respectively). This value will be used as the final exponent if left shifting one place is performed to normalize the coefficient.

Xj + Xk ADDER:*

Refer to Figure 7.5-32 during the following discussion. The feeder registers are located on QA modules and hold the complement of the Xj and Xk exponents. Recall that when adding the complements of numbers, the results is also in complement form. For example, in adding A + B:

TRUE ADD:	ADDING COMPLEMENTS:
A = 0023	$\overline{A}$ = 7754
B = 0042	$\overline{B}$ = 7735
SUM = 0065	$\overline{SUM}$ = 7712
	SUM = 0065

* Note that the Xj + Xk Adder logic is exactly the same as that of the Peripheral Processor A and Q Adders using QA, QB, QC and TJ modules.

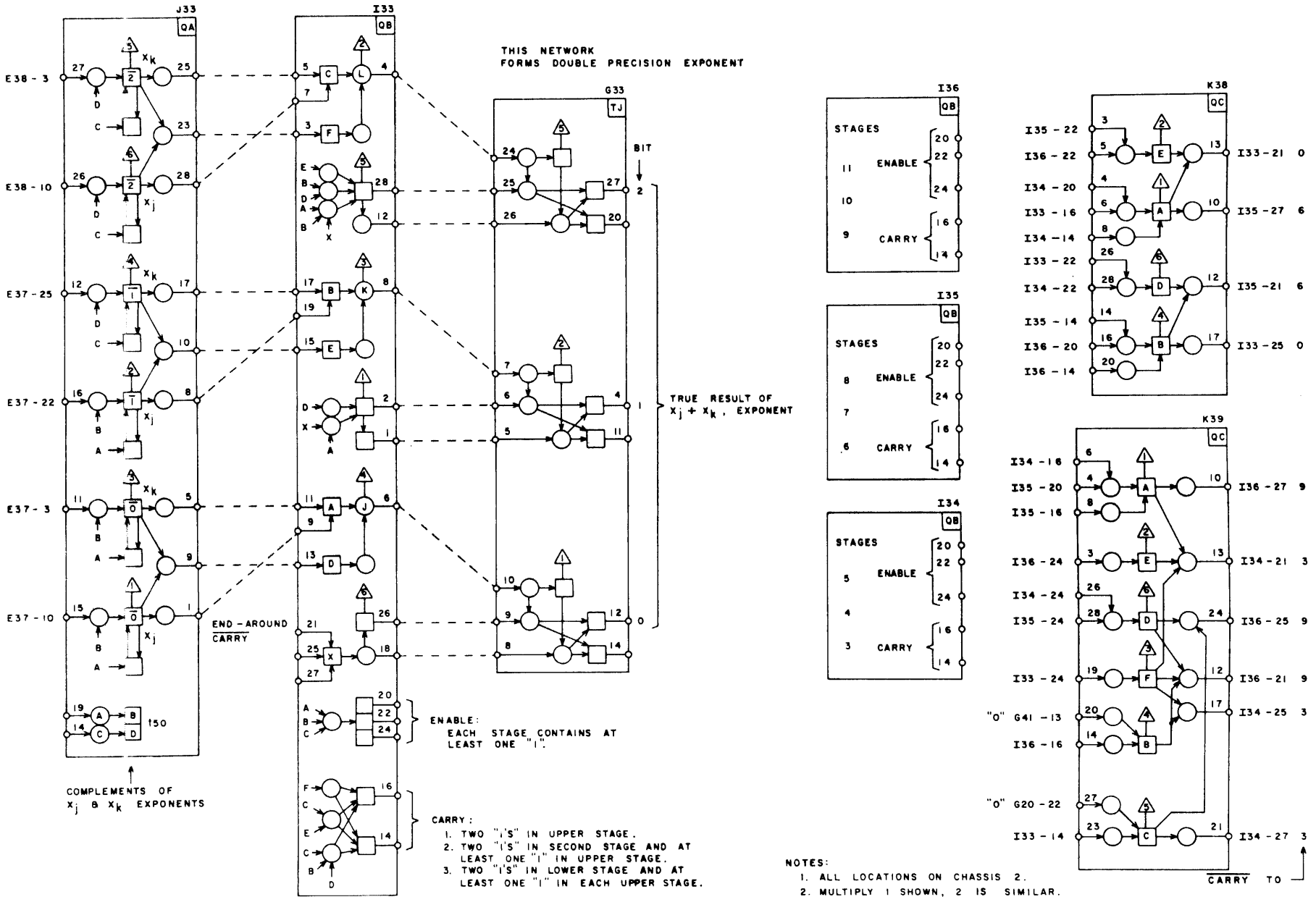

J33
QA
I33
QB
G33
TJ
THIS NETWORK
FORMS DOUBLE PRECISION EXPONENT
E38 - 3
E38 - 10
E37 - 25
E37 - 22
E37 - 3
E37 - 10
BIT
2
1
0
TRUE RESULT OF
$X_j + X_k$, EXPONENT
END - AROUND
CARRY
150
COMPLEMENTS OF
X_j & X_k EXPONENTS
ENABLE:
EACH STAGE CONTAINS AT
LEAST ONE "1".
CARRY:
1. TWO "1'S" IN UPPER STAGE.
2. TWO "1'S" IN SECOND STAGE AND AT
LEAST ONE "1" IN UPPER STAGE.
3. TWO "1'S" IN LOWER STAGE AND AT
LEAST ONE "1" IN EACH UPPER STAGE.
I36
QB
STAGES
11
10
9
ENABLE
CARRY
20
22
24
16
14
I35
QB
STAGES
8
7
6
ENABLE
CARRY
20
22
24
16
14
I34
QB
STAGES
5
4
3
ENABLE
CARRY
20
22
24
16
14
K38
QC
I35 - 22
I36 - 22
I34 - 20
I33 - 16
I34 - 14
I33 - 22
I34 - 22
I35 - 14
I36 - 20
I36 - 14
I33 - 21 0
I35 - 27 6
I35 - 21 6
I33 - 25 0
K39
QC
I34 - 16
I35 - 20
I35 - 16
I36 - 24
I34 - 24
I35 - 24
I33 - 24
"0" G41 - 13
I36 - 16
"0" G20 - 22
I33 - 14
I36 - 27 9
I34 - 21 3
I36 - 25 9
I36 - 21 9
I34 - 25 3
I34 - 27 3
CARRY TO
NOTES:
1. ALL LOCATIONS ON CHASSIS 2.
2. MULTIPLY 1 SHOWN, 2 IS SIMILAR.

Figure 7.5-32

In the true adder:

a) $\text{EQUIVALENCE} \cdot \text{CARRY IN} + \overline{\text{EQUIVALENCE}} \cdot \overline{\text{CARRY IN}} = 1$

b) $\overline{\text{EQUIVALENCE}} \cdot \text{CARRY IN} + \text{EQUIVALENCE} \cdot \overline{\text{CARRY IN}} = 0$

In order to add complements and obtain a true sum, it is necessary to complement the result of the add. In the Xj + Xk adder this is accomplished by reversing the meaning of a carry. A carry will be generated from stage "X" if both the feeder registers contain a "one", or if a carry enters stage X from X-1 and stage X is not a satisfy condition. But, since the feeders contain the complemented values of Xj and Xk a generate condition (1/1) indicates a Satisfy as far as the true values are concerned. In the same manner, a satisfy condition (0/0) in the feeders actually indicates the presence of a carry when referring to true operands. Thus, the definition of a carry has been reversed and causes the result to be complemented "automatically" within the adder. With reference to the true values of Xj and Xk (not the contents of the feeder), the formulas defining sums of 1 and 0 become:

1) $\overline{\text{EQUIVALENCE}} \cdot \text{CARRY IN} + \text{EQUIVALENCE} \cdot \overline{\text{CARRY IN}} = 1$

2) $\text{EQUIVALENCE} \cdot \text{CARRY IN} + \overline{\text{EQUIVALENCE}} \cdot \overline{\text{CARRY IN}} = 0$

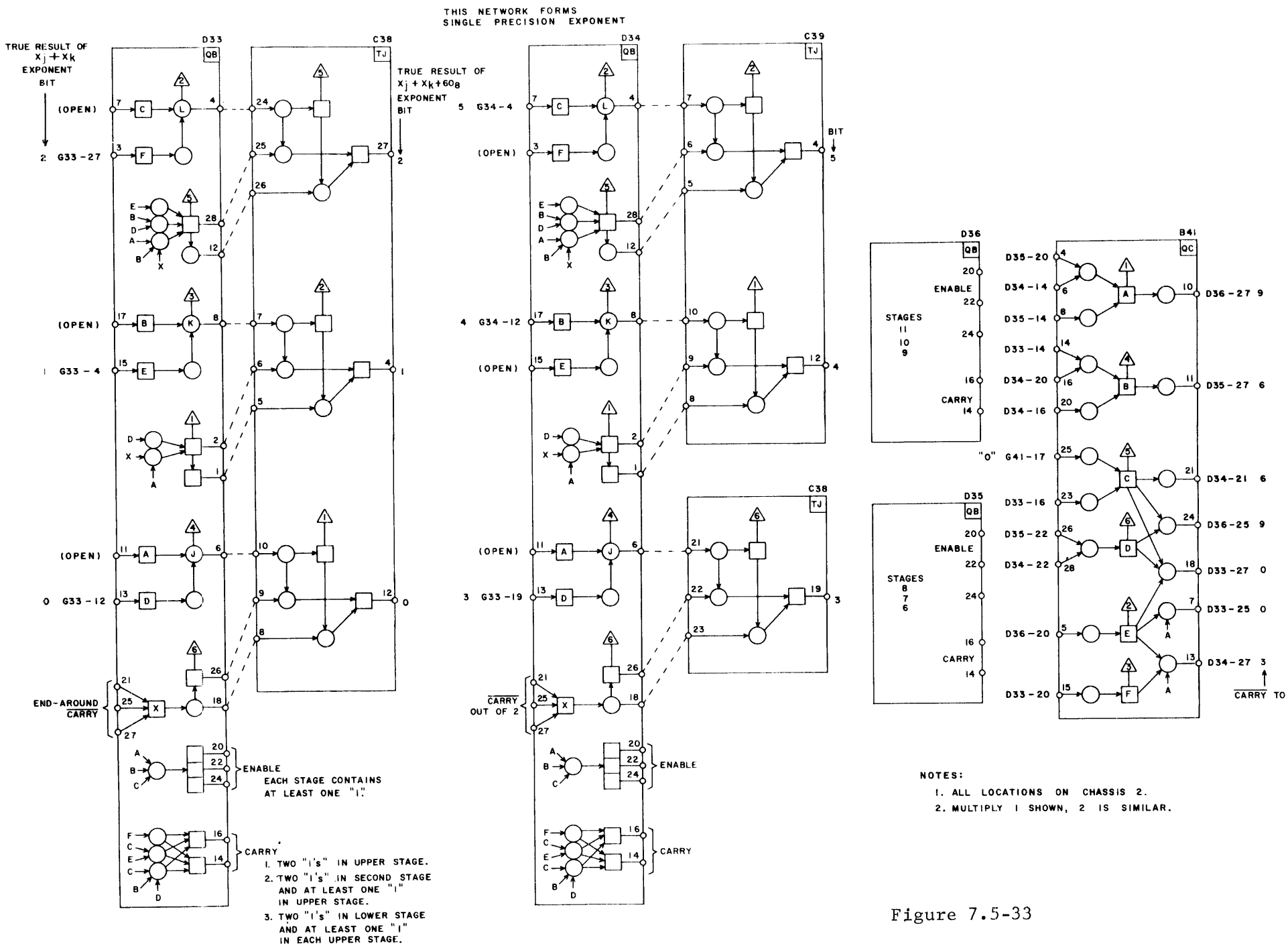
The student is now referred to Figure 7.5-32 where the Xj + Xk Adder is representatively shown. To the left are the QA modules which contain the feeder registers. Next are QB modules which check for stage equivalence and carries entering stages within a 3-bit group. For example, test points 4, 3 and 2 indicate "equivalence" in stages 2^0, 2^1 and 2^2, respectively. Also, test points 6, 1 and 5 indicate

the "carry in" condition to stages 0, 1 and 2 respectively. These conditions are combined on the TJ modules according to formulas #1 and #2 above. To the right are shown more QB modules and QCs which determine which groups have carries in. This determination is made by combining the carry out (generate) and enable conditions of individual stages. The procedure used for carry checks is similar to that used in other 6600 adders and is therefore not belabored. If detailed logic analysis is required, the Chassis #2 wiring tabs should be used.

As indicated, the true value of the result is seen at the output pins of the TJ modules. One of the output pins (for each bit) goes to the second Adder logic where $60_{(8)}$ is added to the sum Xj + Xk. The second output pin is wired to a fan-out which distributes this sum (if in Double Precision mode) to the Decrement, Complement and Error check logic.

Xj + Xk - $60_{(8)}$ ADDER

Refer to Figure 7.5-33 during the following discussion. Since QB and TJ modules are used in this adder, as in the Xj + Xk adder, the concept of the two adders must be the same; in other words, the result is generated by adding complements. No feeder registers are required, since one input is the output of adder #1 and the other input is always $60_{(8)}$ or $000\ 000\ 110\ 000_{(2)}$. To understand the wiring of the inputs to this adder, the values that the QB modules "look for" should be kept in mind. For example, in the Xj + Xk



NOTES:

1. ALL LOCATIONS ON CHASSIS 2.
2. MULTIPLY 1 SHOWN, 2 IS SIMILAR.

Figure 7.5-33

adder the following translations for various QB terms are made (with relation to the feeders):

Term J $\Rightarrow$ EQUIVALENCE (i.e. $Xj \cdot Xk + \overline{Xj} \cdot \overline{Xk}$)
Term A $\Rightarrow$ $\overline{Xj} \cdot \overline{Xk}$ (or, $Xj + Xk$)
Term D $\Rightarrow$ $Xj \cdot Xk$

Since one of the inputs to the $Xj + Xk + 60_{(8)}$ adder is $111\ 111\ 001\ 111_{(2)}$ (the complement of $60_{(8)}$) terms A, B and C on D33 (bits 2^0-2^2) and term A on D34 (bits 2^3-2^5) can be forced to a "1" (since two zeros are never possible in the first four or upper six stages).

On the other hand, terms D, E, and F on D33 and term D on D34 should output a "1" when the corresponding bit of the result, Xj + Xk, is a "zero". This is justified, since if a feeder register were used, it would hold a "1" when the result was actually a "zero" (since it would hold the complement). Because the second input is always $111\ 111\ 001\ 111_{(2)}$ the condition of two ones in bits 0-3 and 6-11 occurs when ever the corresponding bit of the sum, Xj and Xk, is a zero. Hence, the following translations are used:

D33, term D $\Rightarrow$ $\overline{2^0}$
D33, term E $\Rightarrow$ $\overline{2^1}$
D33, term F $\Rightarrow$ 2^2
etc.

Note that two "1"s is also the condition which generates a "carry" (see the inputs to TP1).

The bit 2^4 and 2^5 logic differs somewhat from the other bits since

one input to these stages is always a "zero" (i.e. $60_{(8)}$ complemented). In this case, two "ones" in the feeders is never possible and terms E and F on D34 may be forced to zeros. This is done by leaving the input pins open. Terms B and C are now enabled (pins 19 and 5 are open) and will output a 1 if 2^4 or 2^5 respectively is a "zero". Thus, the outputs of test points 4, 3 and 2 on the QB modules indicate the "Equivalence" condition and the outputs of test points 6, 1 and 5 the "carry in" condition. Carry in checks and final equivalence/ carry summation are identical to the Xj + Xk adder and are, therefore, not discussed further.

MINUS ONE NETWORK

This network, shown in Figure 7.5-34, subtracts one from the output of adder #1 or adder #2 (Double or Single Precision) by adding one to the complement. This is accomplished on GH and IX modules. The inputs to the modules, pins 12, 10 and 8 are the complement of the value to be decremented by one. The true value of the decremented result will be seen at the output, pins 6, 19 and 17.

The following example is used to prove the logic:

Pencil & Paper Method	Logic Method
Xj + Xk = 0572	Xj + Xk = 7205
minus 1 = - 1	plus 1 = + 1
Final exponent = 0571	Sum = 7206
	Sum = 0571

The following translations will occur:

term A = 1 and B = 0
term C = 0 and D = 1
term E = 1 and F = 0
term K = 0 (K ⇒ all 12-bits = zeros)

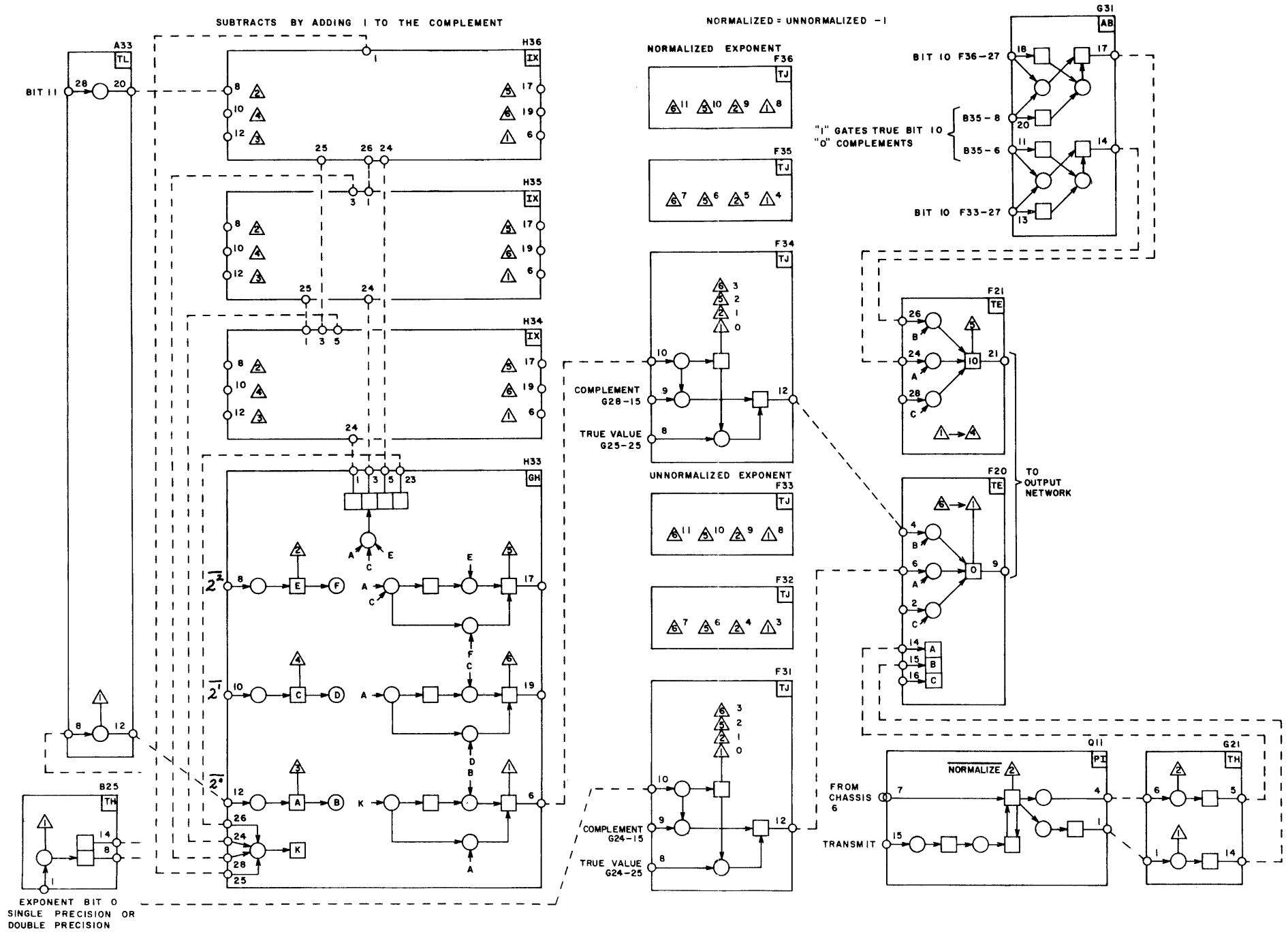

SUBTRACTS BY ADDING 1 TO THE COMPLEMENT
NORMALIZED = UNNORMALIZED −1
BIT 11
A33
TL
H36
IX
H35
IX
H34
IX
H33
GH
NORMALIZED EXPONENT
F36
TJ
F35
TJ
F34
TJ
UNNORMALIZED EXPONENT
F33
TJ
F32
TJ
F31
TJ
COMPLEMENT G28-15
TRUE VALUE G25-25
COMPLEMENT G24-15
TRUE VALUE G24-25
G31
AB
BIT 10 F36-27
"1" GATES TRUE BIT 10
"0" COMPLEMENTS
B35-8
B35-6
BIT 10 F33-27
F21
TE
F20
TE
TO OUTPUT NETWORK
Q11
PI
NORMALIZE
FROM CHASSIS 6
TRANSMIT
G21
TH
B25
TH
EXPONENT BIT 0
SINGLE PRECISION OR
DOUBLE PRECISION

Figure 7.5-34

Translation of test points 5,6 and 1 will yield 0, 0, 1, respectively which is the correct result for the lowest octal digit. The remaining bits work similarly and can be proven with the use of the wire tabs.

The selection of the normalized or unnormalized exponent is determined by the $\overline{\text{NORMALIZE}}$ flip-flop located on Q11. (Chassis 6 will set this flip-flop, via pin 7, if a left shift to normalize is not required). The flip-flop will enable either term A or B on modules F20 and F21. (These contain fan-ins for the 12 result bits and will thereby select the normalized or unnormalized exponent.) Incidentally, the selected exponent will be in the correct form (i.e. true or complemented) as determined by the final sign of the result (Module F34, pins 8 and 9).

7.5.12 EXPONENT TEST RESULT

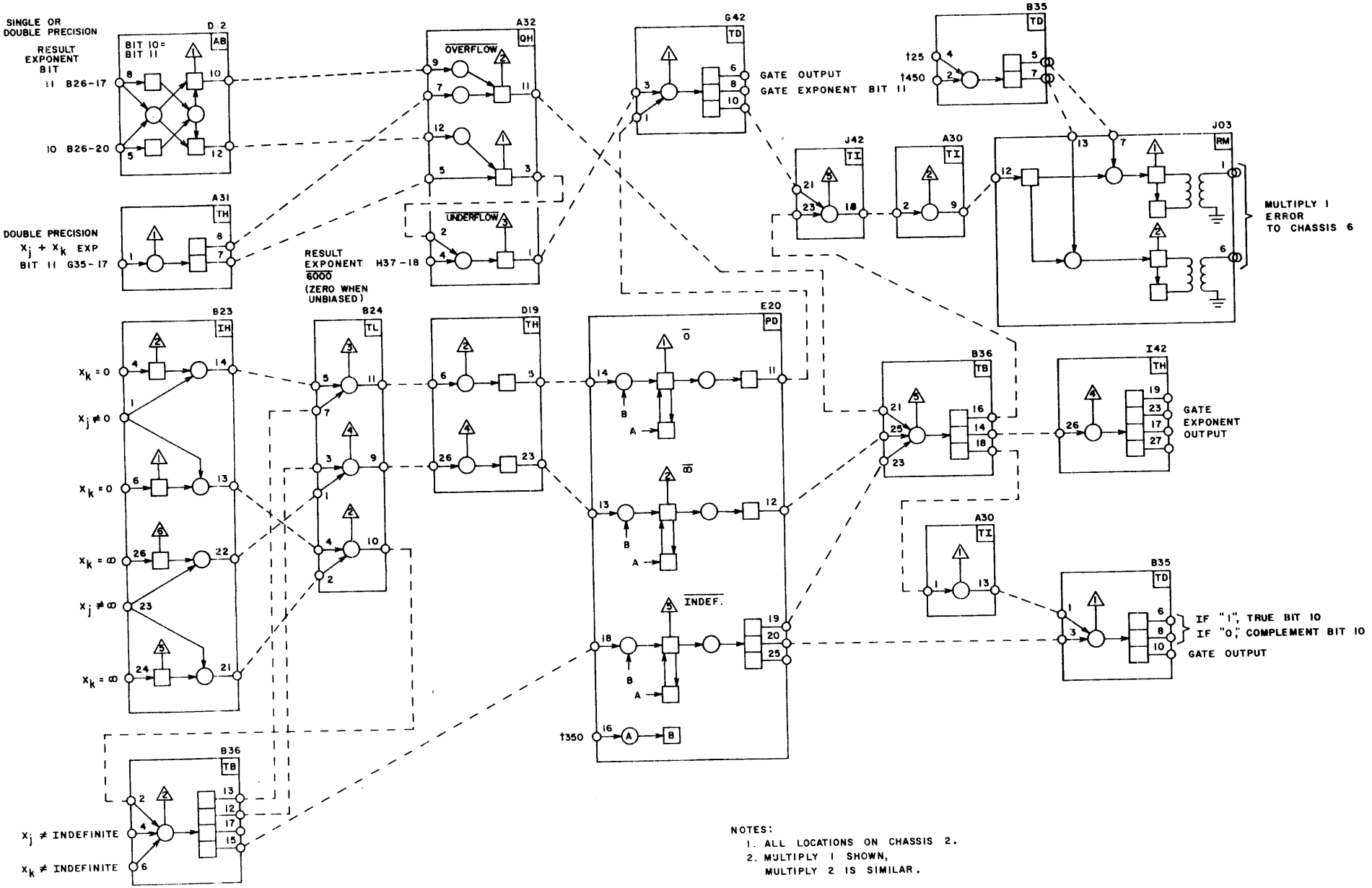
The determination of whether or not an error condition exists is made by checking 1) the exponents of the source operands, Xj and Xk, and 2) the result generated by the addition of the Xj and Xk exponents. Five names are used in discussing the error conditions; three are determined by checking source operands and two by checking the result.

1)	ZERO	exponent of SOURCE operands
2)	INFINITE	
3)	INDEFINITE	
4)	OVERFLOW	exponent of result
5)	UNDERFLOW	

In other words, the Zero, Infinite and Indefinite conditions can be determined before calculating the sum of the exponents, while Overflow and Underflow are generated by the addition of the two exponents. Figure 7.5-35 shows the logic circuitry used for exponent error checking and should be referenced during the following discussion.

The zero condition will occur whenever Xj or Xk, bits 48-59 are zero (i.e. 0000 or 7777) and the second operand is neither indefinite (1777 or 6000) nor infinite (3777 or 4000). These conditions are checked on B24 by TP3. A "zero" on pin 5 indicates that neither Xj nor Xk equals zero. A "zero" on pin 7 indicates that one or the other or both of the operands are indefinite or infinite. A "zero" on either pin forces a "one" out of TP3 and allows E20, TP1 to be set. In Boolean form, the condition, ZERO, implies:

$$(Xj = 0)(Xk \neq \text{indef.})(Xk \neq \text{inf.}) + (Xk = 0)(Xj \neq \text{indef.})(Xj \neq \text{inf.})$$



NOTES:
1. ALL LOCATIONS ON CHASSIS 2.
2. MULTIPLY 1 SHOWN, MULTIPLY 2 IS SIMILAR.

Figure 7.5-35

The INFINITE condition will occur whenever Xj or Xk bits 48-59 have the configuration, 3777 or 4000, and the second operand is neither indefinite nor zero. These conditions are checked on B24 by TP4. A "zero" on pin 1 indicates that neither Xj nor Xk are infinite. A "zero" on pin 3 indicates that one or the other or both operands are indefinite or zero. A "zero" on either pin forces a "one" out of TP4 and allows E20, TP2 to be set. In Boolean form, the condition INFINITE implies:

$$(Xj = \text{inf.})(Xk \neq \text{indef.})(Xk \neq \text{zero}) + (Xk = \text{inf.})(Xj \neq \text{indef.})(Xj \neq \text{zero})$$

The INDEFINITE condition will occur whenever Xj or Xk bits 48-59 have the configuration, 1777 or 6000, or one of the exponents equals zero and the second is infinite. These conditions are checked on B36 by TP2. A "zero" on pin 6 indicates that Xk is indefinite. A "zero" on pin 4 indicates that Xj is indefinite. A "zero" on pin 2 indicates that one operand is zero and the other is infinite. Hence, a "one" at the output (i.e. pin 15) indicates that the Indefinite condition does not exist and allows E20, TP5 to be set. In Boolean form, the condition INDEFINITE implies:

$$(Xj = \text{indef.}) + (Xk = \text{indef.}) + (Xj = 0)(Xk = \quad) + (Xj = \quad)(Xk = 0)$$

The OVERFLOW condition occurs when the sum of Xj + Xk exceeds 3777. This check is made on A32, TP2. A "zero" on pin 9 indicates that bits 10 and 11 of the result are not equivalent (i.e. 1, 0 or 0,1). A "zero" on pin 7 indicates that bit 2^{11} of the result is a "zero".

When both of these inputs are "zero", overflow has occurred. (The upper 2 bits of the unbiased result should be "zeros" for the non-overflow case - see Exponent Formation in Section 7.5.1). If either pin is a "one", the condition, $\overline{\text{overflow}}$, is indicated by a "one on pin 11. In Boolean form, the condition OVERFLOW implies:

$$(2^{10} \neq 2^{11})(\overline{2^{11}})$$

The check for UNDERFLOW is similar to that of overflow, but in this case the upper two bits of the result should both be "ones". The underflow condition is checked by A32, TP1. A "zero" on pin 12 indicates equivalence between bits 10 and 11. A "one" on pin 5 indicates that bit 11 is a one. When pin 12 = "0" and pin 5 = "1", pin 3 = "0" and indicates that underflow has occurred. Pin 3 is wired to pin 2. Pin 4 indicates a second possibility for generating underflow, that is, when the sum of the exponents reaches exactly 0000. In this case, the upper two bits will both be set and TP1 will not indicate underflow. A separate network (C. E. Diagrams, Sheet 163) is used to check for the 6000 condition. The circuit fed by pins 2 and 4 "ORs" these two possible underflow cases. In Boolean form, the condition UNDERFLOW implies:

$$(2^{10} \neq 2^{11})(2^{11}) + \text{Result} = 6000$$

These error conditions are combined to enable sending an error signal to Chassis 6 and to gate the output network. The output translation for B36, pins 14, 16 and 18 is:

$$(\overline{\text{OVERFLOW}})(\overline{\text{INFINITE}})(\overline{\text{INDEFINITE}})$$

This translation is sent, via pin 6, to J42, pin 23 where it is ANDed with pin 21, which translates as:

$$(\overline{\text{ZERO}})(\overline{\text{UNDERFLOW}})$$

The translation of A35, pin 16 is then:

$$(\overline{\text{ZERO}})(\overline{\text{UNDERFLOW}})(\overline{\text{OVERFLOW}})(\overline{\text{INFINITE}})(\overline{\text{INDEFINITE}})$$

This condition disables sending a Multiply 1 error signal to Chassis 6. The opposite condition, i. e.

$$\text{ZERO} + \text{UNDERFLOW} + \text{OVERFLOW} + \text{INFINITE} + \text{INDEFINITE}$$

causes the error signal to be sent.

The output of B36 also enables the "gate output" signal which enables the exponent to the output network (see Section 7.5.13).

7.5.13 EXPONENT OUTPUT NETWORK

Figure 7.5-36 shows the logic for the exponent output network. This circuitry determines when to force the Non-Standard operand bit configurations (0000, 3777, 4000 & 1777) and when to enable the sum of the exponents to the transmitters (RM modules).

To enable the sum of the exponents to the transmitters, term B on the RM modules must be a "one". This is obtained as follows. H37, pin 13 must be a "one", this implier:

$$\text{LIKE SIGNS} + \text{INDEFINITE} + (\overline{\text{INFINITE}})(\overline{\text{OVERFLOW}})$$

On D16, this condition is ANDed with pin 1 to yield the following translation for pin 19:

$$(\overline{\text{ZERO}})(\overline{\text{UNDERFLOW}})(\text{LIKE} + \text{INDEFINITE} + \overline{\text{INFINITE}} \cdot \overline{\text{OVERFLOW}})$$

This translation is combined with a t00 on H37 and with the "Transmit" signal on Ø09 to yield the following translation for a "zero" from pin 11:

$$(\text{t00})(\text{Transmit})(\overline{\text{ZERO}})(\overline{\text{UNDERFLOW}})(\text{LIKE} + \text{INDEFINITE} + \overline{\text{INFINITE}} \cdot \overline{\text{OVERFLOW}})$$

This term, via H25, causes term B to be a "1" and enables the sum of the exponents to the transmitters. By complementing the above formula, the cases when the transmitters are disabled can be seen.

$$\overline{\text{t00}} + \overline{\text{Transmit}} + \text{ZERO} + \text{Underflow} + \text{Unlike} \cdot \overline{\text{INDEFINITE}} \cdot (\text{Infinite} + \text{Overflow})$$

Study of the formulas will reveal that the transmitters are disabled when the following Non-Standard operands occur:

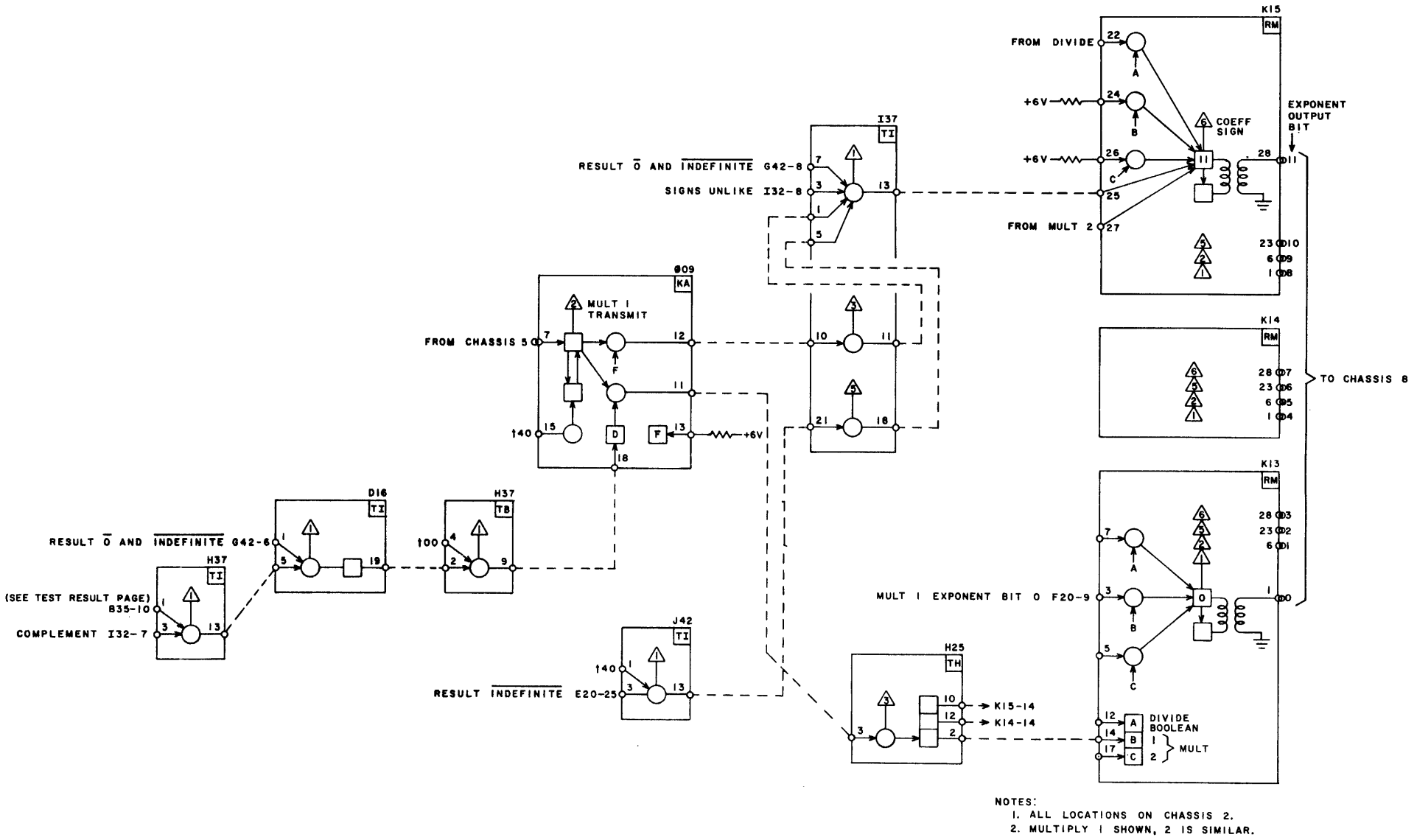

K15
RM
FROM DIVIDE
+6V
+6V
COEFF SIGN
EXPONENT OUTPUT BIT
I37
TI
RESULT $\bar{0}$ AND $\overline{\text{INDEFINITE}}$ G42-8
SIGNS UNLIKE I32-8
FROM MULT 2
009
KA
MULT 1 TRANSMIT
FROM CHASSIS 5
140
+6V
K14
RM
TO CHASSIS 8
K13
RM
MULT 1 EXPONENT BIT 0 F20-9
DIVIDE BOOLEAN
MULT
D16
TI
H37
TB
RESULT $\bar{0}$ AND $\overline{\text{INDEFINITE}}$ G42-6
100
H37
TI
(SEE TEST RESULT PAGE) B35-10
COMPLEMENT I32-7
J42
TI
140
RESULT $\overline{\text{INDEFINITE}}$ E20-25
H25
TH
K15-14
K14-14
NOTES:
1. ALL LOCATIONS ON CHASSIS 2.
2. MULTIPLY 1 SHOWN, 2 IS SIMILAR.

Figure 7.5-36

1) Positive or negative zero or underflow (0000 or 7777)
2) Negative infinity or overflow (4000)

They are enabled when the following Non-standard operands occur:

1) Positive infinity or overflow (3777)
2) Positive or negative indefinite (1777)

This proves that only the following Non-standard operands can be generated by the Multiply Unit:

1) 0000X ——— X (for ± zero or underflow)
2) 3777X ——— X (for + infinity or overflow)
3) 4000X ——— X (for - infinity or overflow)
4) 1777X ——— X (for ± indefinite)

Since the transmitters are enabled for positive infinity and negative or positive indefinite conditions, the sum of the exponents must be disabled and "ones" must be forced into the transmitters for bits 0-9. This occurs via module I42, TP4 whose output is a "one" when Indefinite, Infinite or Overflow occur (Figure 7.5-35). This condition is fed to modules G25 and G24, pins 5 and 16 and forces both the True and Complement gates to be a "one" (pins 25 and 15). These conditions force both the true and complemented values out of the TJ modules (adder outputs - see Figure 7.5-34). In other words, all "ones" are forced out of the TJs and to the output network fan-ins (TE modules). This results in "ones" being fed to the transmitter input pins (i.e. K13, pin 9 in Figure 7.5-36) and, consequently, "ones" are transmitted on the data trunk.

Bit 2^{10} and 2^{11} are handled separately. Bit 2^{10} is taken from the adder output network (TJ modules) but may be gated to the transmitters

in True or Complement form. (See Figure 7.5-34.) The true form is used if the indefinite condition does not exist AND infinite or overflow does exist. The complement of bit 10 is used if indefinite is present or infinite and overflow are not present. In summary, Bit 10 equals:

$$\text{TRUE if } (\overline{\text{INDEFINITE}})(\text{INFINITE} + \text{OVERFLOW})$$
$$\text{FALSE if INDEFINITE} + (\overline{\text{INFINITE}})(\overline{\text{OVERFLOW}})$$

The logic which determines whether 2^{11} is a "zero" or a "one" is shown in Figure 7.5-36, module I37. When all inputs to TP1 are "ones", a "one" will be transmitted in Bit 2^{11}. The input formula is:

$$(\text{t50})(\overline{\text{ZERO}})(\overline{\text{UNDERFLOW}})(\text{UNLIKE})(\overline{\text{INDEFINITE}})(\text{XMIT})$$

The complement of the above formula will indicate the cases when 2^{11} is a "zero":

$$\overline{\text{t50}} + \text{ZERO} + \text{UNDERFLOW} + \text{LIKE} + \text{INDEFINITE} + \overline{\text{XMIT}}$$

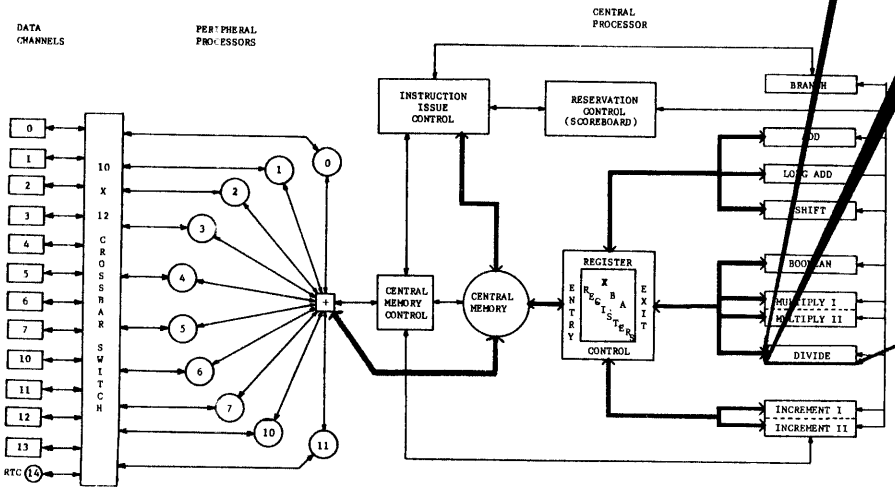
The above formulas can be justified by relating them to all possible conditions for bit 2^{11}.

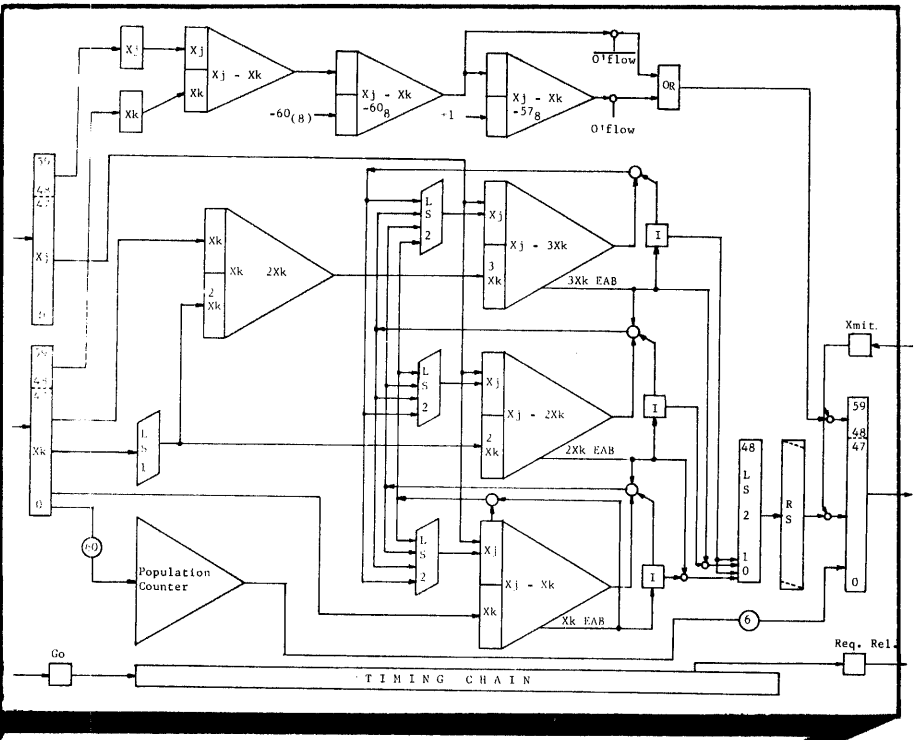
SECTION 7.6

DIVIDE

FUNCTIONAL UNIT

DIVIDE FUNCTIONAL UNIT





7.6.1. INTRODUCTION

GENERAL: The Divide functional unit performs single precision, rounded or unrounded floating point division of two operands (Xj/Xk) and will sum the number of "ones" in a selected X register (Xk). If division is performed, the functional unit time is 2.9 micro seconds. Counting the number of ones requires 800 nanoseconds.

The Divide Unit is located on Data Trunk #2 along with the Multiply 1, Multiply 2 and Boolean units. Divide holds first priority in reading operands and second priority in storing results. The unit is physically located on chassis 2, but a portion of the data transfers is via chassis 6. Since the Multiply coefficient logic is on chassis 6, bits 0-47 and 59 of the source operands are gated from Exit Control to chassis 6 and them to chassis 2. Bits 48-59 are sent directly from Exit Control to chassis 2. Figure 7.6.-1 illustrates the data flow, which is manipulated similarly for result operands.

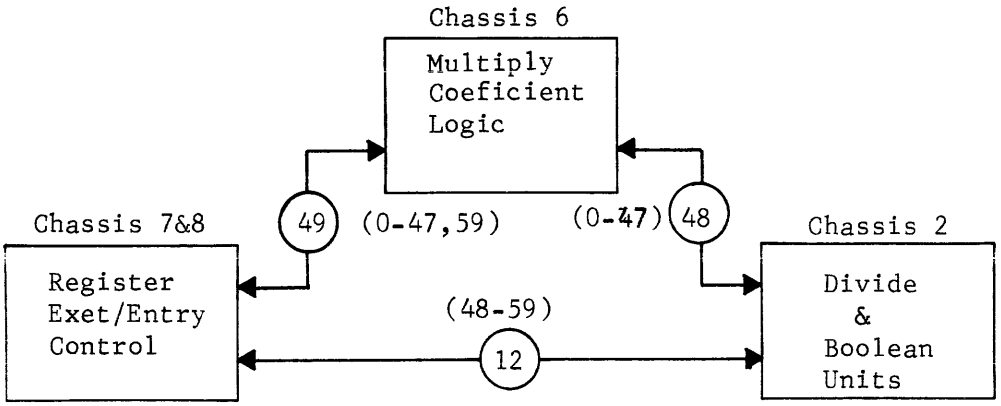


Figure 7.6-1

THE DIVIDE PROCESS (COEFFICIENT): Coefficient division of the two operands (Xj/Xk) is performed using three subtraction networks. The orginal dividend, Xj, is gated directly from the chassis 2 input register into each of the three subtractors. The divisor Xk, is gated into a holding register from which three values are formed: 1, 2 and 3 times Xk (here after referred to as 1X, 2X and 3X respectively). These quantities are gated into the 3 subtraction networks where they are held for the duration of the operation.

The quotient is formed 2 bits at a time by trial subtractions fo the 3 multiples of Xj from the partial dividend (initially, the dividend Xj). The largest multiple which subtracts without causing an end-around borrow determines the selection of the two bits (1.e 01, 10 or 11). If end-around borrows occur in all three subtractions, the two bits are zero.

The quantity that results from the subtraction of the largest usable multiple of Xk from the partial dividend is left shifted 2 places and re-inserted into the dividend registers of the three subtraction networks. This occurs for each of the 24 iterations of the divide operation.

A 50-bit quotient is formed since 2 bits of the quotient are selected before the first and after the 24th iteration. The high order bit (2^{49}) of the final quotient is never used due to the fractional nature of the coefficient arithmetic.

The lower 48-bits (2^0-2^{47}) of the final, 50-bit quotient will be selected as the result if bit 2^{48} does not equal a "one". If 2^{48} is a "one", bits 2^1 through 2^{48} are taken as the final result. Note, that bit 2^{49} of the 50-bit result can never be retrieved, and it, therefore, should always be a "zero". If it is a "one", the most significant bit of the quotient is lost and the final quotient sent to Xi will be meaningless. Thus, the ratio between Xj and Xk must always be less than 2 to 1, since this will cause the upper two bits of the 50-bit result to be either $00_{(2)}$ or $01_{(2)}$.*

The following example will be used to illustrate the coefficient divide method. To simplify the division process, 12 bit coefficients are used, rather than 48.

Xj = 7604

Xk = 5213

```
                    1.3613
Xj/Xk =   5213   7604.0000
                 5213
                 23710
                 17641
                  40470
                  37502
                    7660
                    5213
                    24450
                    17641
                     4607
```

* To insure that the ratio of the Xj and Xk coefficients is less than 2 to 1, it is suggested that the operands used with the 44 and 45 opcodes always be normalized. This will cause the final coefficient to be in the range, 0.0 —— 0 through 1.7 —— $7_{(8)}$.

The machine method follows:

1) Form 1X, 2X, 3X:

1X = 101 010 001 011

2X = 001 010 100 010 110

3X = 001 111 110 100 001

2) Perfom first trial subtraction:

111 110 000 100 = Xj

101 010 001 011 = 1X

010 011 111 001 = Partial Dividend

Since 1X could be subtracted, the upper 2 quotient bits are "01"

The partial quotient is therefore:

01. XXX XXX XXX XXX

3) Left shift the partial dividend and perform the second trial subtraction:

001 001 111 100 100 = Partial Dividend Left 2

101 010 001 011 = 1X

100 101 011 001 = New Partial Dividend

Since 1X could be subtracted, the next 2 quotient bits are "01"

The partial quotient is therefore:

01. 01X XXX XXX XXX

4) Left Shift the partial dividend and perform the trial subtraction:

010 010 101 100 100 = Partial Dividend Left 2

001 111 110 100 001 − 3X

010 111 000 011 = New Partial Dividend

Since 3X could be subtracted, the next 2 quotient bits are "11".

5) Left shift the partial dividend and perform the fourth trial subtraction:

001 011 100 001 100 = Partial Dividend Left 2

<u>001 010 100 010 110</u> = 2X

111 110 110 = New Partial Dividend

Since 2X could be subtracted, the next 2 quotient bits are "10". The partial quotitent is therefore:

01. 011 <u>110</u> XXX XXX

6) Left shift the partial dividend and perform the fifth trial subtraction:

000 011 111 011 000 = Partial Dividend Left 2

Since subtraction of 1X, 2X or 3X all cause end around borrows, the next two quotient bits are "00".

7) Left shift the partial dividend and perform the sixth trial subtraction:

001 111 101 100 000 = Partial Dividend Left 2

<u>001 010 100 010 110</u> = 2X

101 001 001 010 = New Partial Dividend

Since 2X could be subtracted, the next 2 quotient bits are "10". The partial quotient is therefore:

01. 011 110 001 0XX

8) Left shift the partial dividend and perform the seventh trial subtraction:

001 111 101 100 000 = Partial Dividend Left 2

<u>001 010 100 010 110</u> = 3X

100 110 000 111

Since 3X could be subtracted, the next 2 quotient bits are "11". The quotient is therefore:

01. 011 110 001 011 or

1. 3 6 1 3 $_{(8)}$ which checks with the original long division method.

Since the bit immediately to the right of the binary point is a "1", (using 48-bit coefficients, this would be 2^{48}) the quotient will be right shifted one place (normalized) before packing in floating point format. This step yields:

00. 101 111 000 101 1 or .5 705$_{(8)}$.

Note that seven trial subtractions and six left shifts occurred while processing the 12-bit operands. Similarly, twenty-five trial subtractions and 24 left shifts occur when processing 48-bit coefficients. This indicates how the 50-bit quotient is generated.

EXPONENT FORMATION:

At this point, the positional value of the original and final coefficients must be considered. The source operands used by the divide unit are, of course, in standard floating point format;* that is, a negative or positive 48-bit integer coefficient multiplied by 2 raised to an exponent in the range, $\pm$ 1777$_{(8)}$.

---

* See Appendix A for a detailed discussion of 6000 Series floating point operations.

On the other hand, the quotient generated by the divide aritmetic is a fractional value. Packing this fractional result in standard floating point format effectively moves the binary point 48 places to the right, which has the effect of increasing the quotient magnitude by 2^{48}. To compensate for the increase of coefficient magnitude, the final exponent (the difference of exponents, Xj −Xk) is decremented by $48_{(10)}$ $(60_{(8)})$.

Recall that if the quotient has a value of 1 or greater (1.e. 1.X — X) it is right shifted one place to yield a normalized pure fraction (1.e. 0.4X — X). This effectively decreases the quotient magnitude by 2; and a corresponding increase of the final exponent is required.

It can be concluded, from the above discussion, that three adders are required for the formation of the final exponent. The first will form the algebraic difference of the Xj and Xk exponents, specifically, Xj −Xk. The second subtracts the value, $60_{(8)}$, from the result of the first, Xj −Xk. It therefore forms $Xj - Xk - 60_{(8)}$. This value will be used as the final exponent if quotient overflow (1.e. 2^{48} of the quotient is not a "one") does not occur. The subtraction of $60_{(8)}$ is required to compensate for packing the fractional quotient into integer form. The third adder adds one to the result of the second to form, $Xj - Xk - 57_{(8)}$. This value is used as the final exponent if quotient overflow does occur, in which case a right shift one place makes the quotient a pure fraction.

Figure 7.6-2 represents the exponent adders in block form:

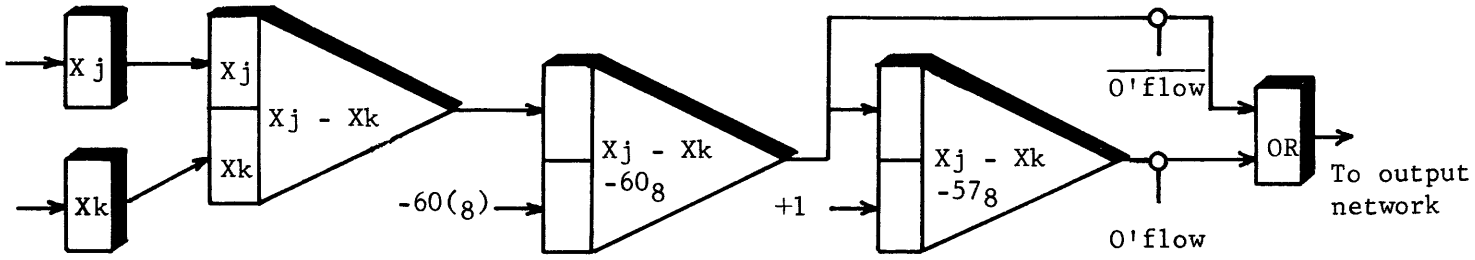


Figure 7.6-2

In forming the difference of the Xj and Xk exponents the following sequence occurs. So that all cases of negative and positive exponents are illustrated, the following numerical examples will be used:

Xj	=	+75	+75	−75 −75
Xk	=	+10	−10	+10 −10
Xj − Xk	=	+65	+105	−105 −65

STEP 1 Obtain the true value of the exponents. This is accomplished by complementing the upper 12-bits if bit 59 is a "1" (negative coefficient). If bit 59 is a "0", the exponent is already in true form. Hence the true values of the exponents in packed form are:

2075	2075	1702	1702
2010	1767	2010	1767

STEP 2 Unpack the exponents by:

a) Extending bit 58 into bit 59 of the feeders:

6075 6075 1702 1702

6010 1767 6010 1767

b) Complementing the exponent magnitude bits into the feeder registers. The feeders now hold the complemented, unbiased exponents:

7702 7702 0075 0075

7767 0010 7767 0010

STEP 3 Subtract. The results are:

7712 7672 0105 0065

STEP 4 Complement the result (since the exponents are in complement form in the feeders):

0065 0105 7672 7712

STEP 5 Set or clear bit 59 according to the following rules:

SET BIT 59 if:

a) The result is not zero (i.e. 0000)

b) AND the result is not indefinite (i.e. 1777)

c) AND underflow did not occur (i.e. 0000)

d) AND the final sign of the coefficient is negative. This is determined by the following rules:

1) like signs $\Longrightarrow$ positive result

2) unlike signs $\Longrightarrow$ negative result

e) AND a "Transmit" was received from the scoreboard.

CLEAR BIT 59 if the above conditions are not met.

STEP 6 Set or clear bit 58 according to the following conditions:

Set if:

$$[(\overline{\text{Indefinite}})(\overline{\text{Underflow}})]\,[(\text{Unlike})(\text{Negative}) + (\text{Like})(\text{Positive} + \text{Overflow})]$$

Clear if:

$$\text{Indefinite} + \text{Underflow} + [\overline{\text{Underflow}}]\,[\overline{\text{Indefinite}}]\,[(\text{like})(\text{Negative}) + (\text{Unlike})(\text{Overflow} + \text{Positive} \cdot \overline{\text{Overflow}})]$$

The following table shows all possible uses for bits 48-59 of the result and should clarify the conditions for setting and clearing bits 58 and 59.

RESULT	CONDITIONS	59	58
0000	U'flow	0	0
0001 - 1776	$(\overline{\text{U'flow}})(\overline{\text{Ind}})(\text{Like})(\text{Neg})$	0	0
1777	Indefinite	0	0
2000 - 3776	$(\overline{\text{U'flow}})(\overline{\text{Ind}})(\text{Like})(\text{Pos})$	0	1
3777	$(\overline{\text{U'flow}})(\overline{\text{Ind}})(\text{Like})(\text{O'flow})$	0	1
4000	$(\overline{\text{U'flow}})(\overline{\text{Ind}})(\text{Unlike})(\text{O'flow})$	1	0
4001 - 5777	$(\overline{\text{U'flow}})(\overline{\text{Ind}})(\text{Unlike})(\text{Pos})$	1	0
6000	Negative Indefinite is not possible	X	X
6001 - 7776	$(\overline{\text{U'flow}})(\overline{\text{Ind}})(\text{Unlike})(\text{Neg})$	1	1

These cases are explained in more detail in section 7.6-9.

7.6.2 INSTRUCTION LIST/DATA FLOW

The following instructions will select the Divide functional unit. The terms in parenthesis are the ASCENT symbolic codes used in assembler coding. Data flow can be followed in Figure 7.6 - 3.

44 FLOATING DIVIDE Xj by Xk to Xi (FXi = Xj/Xk)

DEFINITION: This instruction divides two normalized floating point quantities obtained from operand registers Xj (dividend) and Xk (divisor) and packs the quotient in operand register Xi.*

The exponent of the result in a no-overflow case is the difference of the dividend and divisor exponents minus $48_{(10)}$.

A one bit overflow is compensated for by adjusting the exponent and right shifting the quotient one place. In this case, the exponent is the difference of the dividend and divisor exponents (Xj - Xk) minus 47.

DATA FLOW: The Xj and Xk operands are sent to chassis 2, where exponent and coefficient manipulations take place. The base exponent is formed by subtracting the exponent of Xk from that of Xj. The quantity $60_{(8)}$ is subtracted from the base address by a second adder. This has the effect of moving the binary point of the coefficient 48 places to the right. Since the quotient

---

*The result is a normalized quantity when both the divident and the divisor are normalized. Note that the machine makes no note of divide faults, i.e., when the absolute value of the coefficient of the dividend $\geq$ two times the absolute value of the coefficient of the divisor. To avoid possible incorrect results from using unnormalized operands, the operands in this instruction should be normalized.

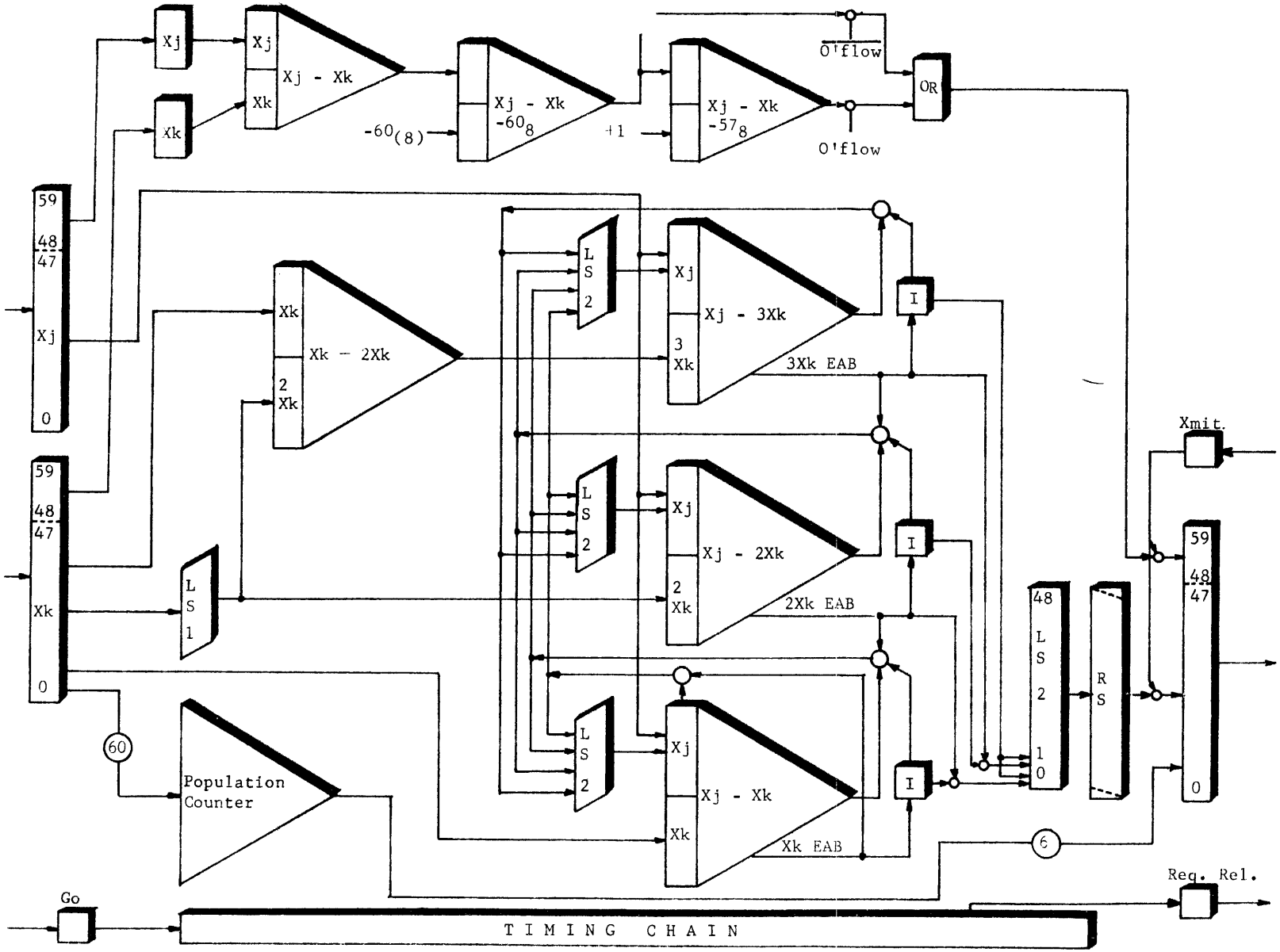

Xj
Xk
Xj - Xk
-60(8)
Xj - Xk
-60₈
+1
Xj - Xk
-57₈
O'flow
O'flow
OR
59
48
47
Xj
0
Xk
Xk − 2Xk
2
Xk
L
S
2
Xj
Xj - 3Xk
3
Xk
3Xk EAB
I
Xmit.
59
48
47
Xk
0
L
S
1
L
S
2
Xj
Xj - 2Xk
2
Xk
2Xk EAB
I
48
L
S
2
1
0
R
S
60
Population
Counter
L
S
2
Xj
Xj - Xk
Xk
Xk EAB
I
6
Req. Rel.
Go
T I M I N G C H A I N

Figure 7.6-3

produced by the coefficient logic is a fraction, it is necessary to right shift the binary point when packing in the standard floating point format.

A third adder adds one to the quantity, Xj - Xk - $60_{(8)}$. This value, Xj - X6 - $57_{(8)}$, is used as the final exponent if a right shift is required to resolve coefficient overflow (i.e., bit 2^{48} of the quotient is a "one"). If the right shift is not required to bring the coefficient in range, the value Xj - Xk - $60_{(8)}$ will be selected as the final exponent.

The coefficient is formed by repeatedly subtracting a multiple of the divisor (1, 2 or 3Xk) from the partial dividend and setting the quotient bits (00, 01, 10, or $11_{(2)}$) depending upon the largest multiple that could be subtracted without generating an End Around Borrow. Twenty-four such iterations are required to generate the 50-bit quotient. (Two bits are selected before the first and after the twenty-fourth iterations.) This method is explained in detail in Section 7.6.1.

45 ROUND FLOATING DIVIDE Xj by Xk to Xi (RXi = Xj/Xk)

DEFINITION: This instruction divides the floating quantity from operand register j (dividend) by the floating point quantity from operand register Xk (divisor) and packs the round quotient in operand register Xi.* Rounding is accomplished by adding one-third

---

*The footnote for the 44 opcode also applies to the 45 opcode.

during the division process. In effect, the quantity "2525.... 25_8" resides immediately to the right of the dividend binary point prior to starting the divide operation. On the first iteration, a "1" is added to the least significant bit of the dividend. After each iteration (subtraction of divisor from partial dividend) a two-place left shift occurs and a "1" is again added to the least significant bit of the partial dividend. Thus, successive iterations gradually bring in the one-third round "quantity" ($25....25_8$).

The result in a no-overflow case is the difference of the dividend and divisor exponents (Xj - Xk) minus $48_{(10)}$.

A one-bit overflow is compensated for by adjusting the exponent and right shifting the quotient one place. In this case, the exponent is the difference of the dividend and divisor exponents minus $47_{(10)}$.

DATA FLOW: The data flow is the same as for the 44 instruction with the exception of the rounding operation. The one-third round is accomplished by effectively adding the value, 2525.... $25_{(8)}$, to the dividend during the divide step. This is actually accomplished by holding "1" inputs on bit 0 of the dividend registers in each of the three subtraction networks through all 24 iterations. Since the dividends are left shifted 2 places on each iteration the value, $2525....25_{(8)}$, is effectively added to Xj. The 60-bit floating point quotient is sent to Xi upon receipt of the "Transmit" signal from the scoreboard.

47 COUNT THE NUMBERS OF ONES in Xk to Xi (CXi = Xk)

DEFINITION: This instruction counts the number of ones in operand register Xk and stores the count in the lower 6 bits of register Xi. Bits 6 through 59 are cleared to zero.

DATA FLOW: The operand, Xk is sent to the input register and then to the Xk feeder register. A static network called the "Population Counter" counts the number of "ones" in the 60-bit operand and generates a 6-bit sum of ones which is transferred to Xi upon receipt of the "Transmit" signal from the scoreboard.

7.6.3 MODE BITS

Two mode bits exist in the Divide functional unit which enable the logic to distinguish the three Divide opcodes:

1) Floating Divide of Xj by Xk to Xi.

2) Round Floating Divide of Xj by Xk to Xi.

3) Count the number of ones in Xk.

The mode bits are named:

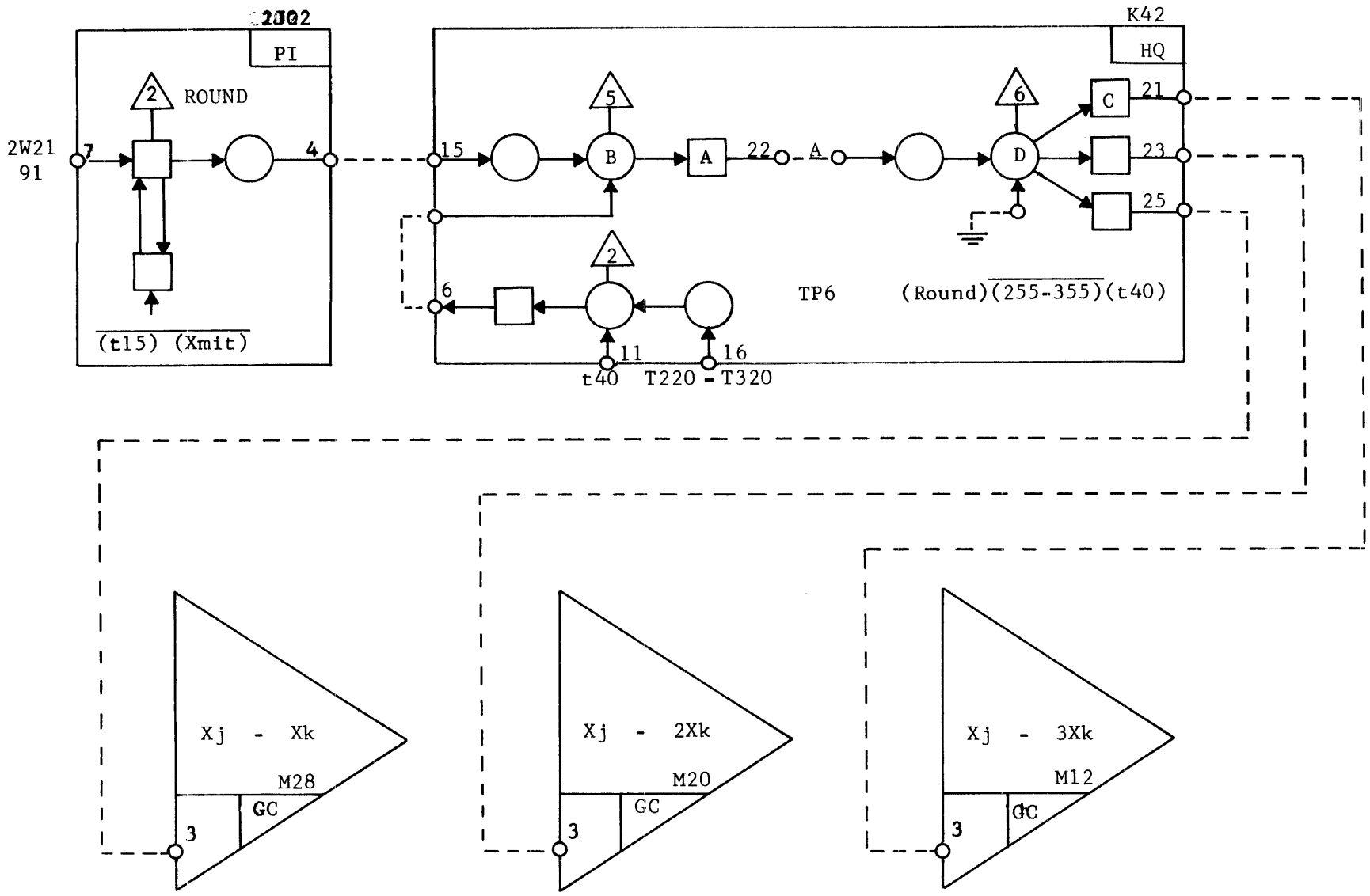
1) Round

2) Population Count

When neither mode bit is sent from chassis 5 and the Divide unit is selected, a Single Precision, Unrounded, Floating point division of Xj by Xk occurs. Twenty-four iterations will occur, as described in Section 7.6.1, in generating an unrounded, 60-bit, floating point quotient which is stored in X register i. Twenty-nine minor cycles are required for the execution of this opcode.

If the "Round" mode bit is sent to chassis 2 upon issuing a Divide opcode, a Single Precision, rounded, floating point division of Xj by Xk occurs. As with Unrounded divide, twenty-four iterations occur and twenty-nine minor cycles are required for the execution of this opcode. A rounded, 60-bit, floating point quotient is generated and stored in X register i.

Figure 7.6-4 shows the "Round" mode bit logic. The catching flip-flop which is set when a 45 opcode is issued to the scoreboard. The flip-flop is cleared when a "Transmit" is received at the end of the divide step. It is therefore, set for at least 2.9 microseconds (no second or third order conflicts); in any case, for the duration



ROUND MODE BIT

Figure 7.6-4

of the divide operation. On K42, test point 5, the condition, "Round", is ANDed with the conditions, "$\overline{(t235 - 335)}(t40)$". Thus, the translation for the output of Test Point 6 is:

$$(ROUND)\overline{(t255 - 355)}(t40)$$

This will cause "zeros" out of K42, pins 21, 23 and 25 which will directly set the 2^0 Xj flip-flops of the three coefficient subtractors. Note that setting the flip-flops is disabled from t255 to 355 of the divide step. This minor cycle corresponds to the first iteration of the divide sequence, during which setting the lowest bit could distort the initial value of Xj (specifically, if bit 2^0 of Xj is a "zero"). In all subsequent iterations, 2^0 will be set. Since the value in the Xj feeder register is left shifted two places each iteration, the end effect will be to add the binary configuration, 010101.....010101 or 1/3, to the initial dividend, Xj. This will cause the final quotient to be rounded up by 1 or 2, depending upon the magnitudes of the divisor and dividend. This rounding method is not completely compatible with the rounding conventions of mathematics and will, on occasion, introduce a small error in the quotient. This method was selected to decrease the execution time of the divide operation and when a number of related rounded divisions and multiplications are programmed, the error introduced in a problem is negligible.

When a "Population Count" mode bit is sent to the Divide unit, the normal divide operation, Xj/Xk, is disabled. Instead, the "ones" in the 60 bit operand, Xk, are counted and the sum of ones is gated to X register i (lower 2 octals).

Figure 7.6 -5 shows a portion of the "Population Count" mode bit logic. The mode bit is received on J02, which has two outputs. Pin 13 is ANDed, on J22, with t510 - 610 of the divide chain to enable the sending of "Request Release" to the scoreboard from E01, Test Point 1. Note the second possibility for "Request Release", entering pin 10 of J22, which is used during normal divide operations.

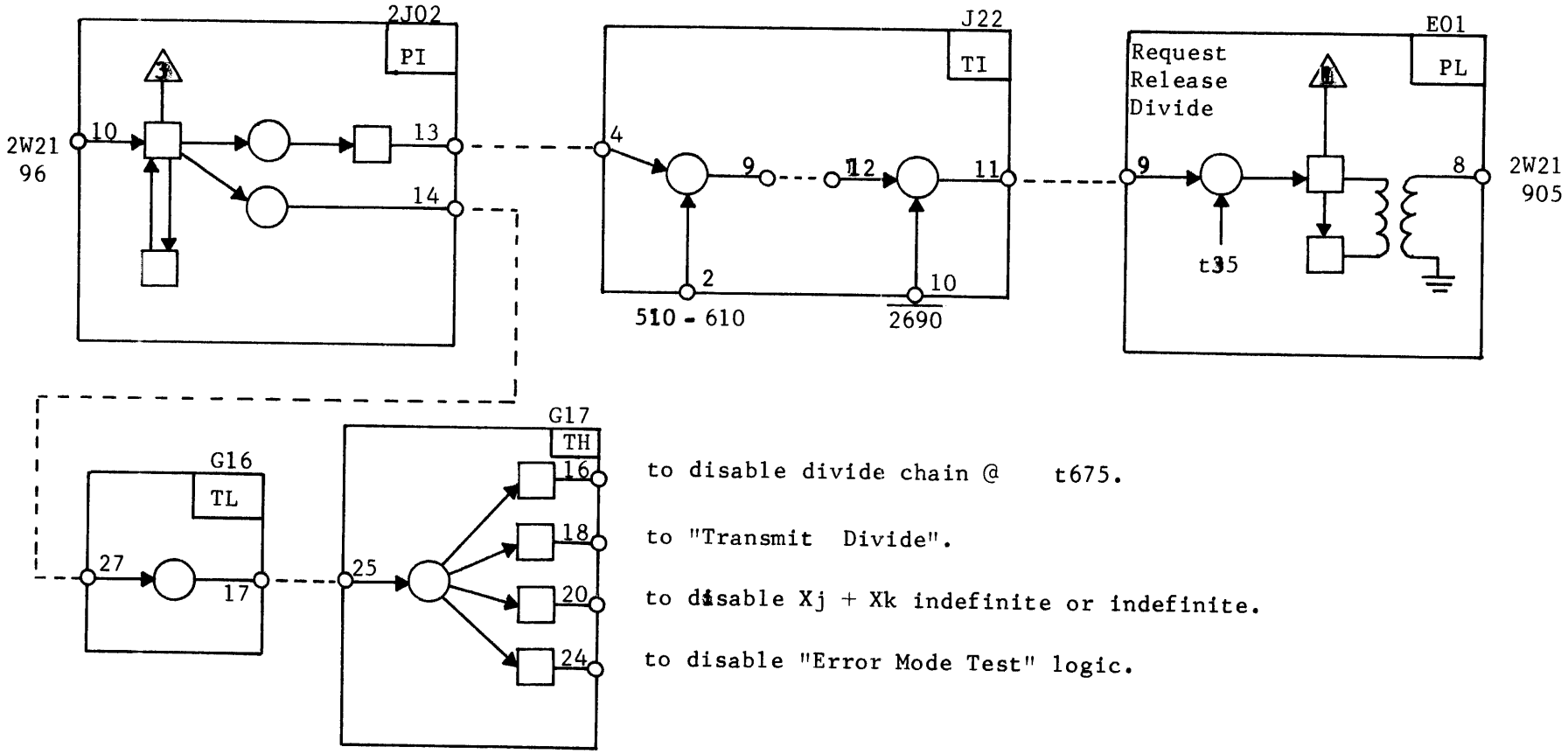
The pin 14 output of J02 is sent, via G16, to G17 where the signal is fanned out to:

1) Disable the divide timing chain (via G17, pin 16), at t675. This accomplished by forcing a "zero" on the clear/set term of the t725 - 825 flip-flop in the timing chain (2D21, TP3 is disabled when pin 9 is a "zero") and thereby disabling the setting of that flip-flop. Hence, all subsequent flip-flops remain cleared.

2) Enable the transmission of the "Pop. Count" to the register chassis (via G17, pin 18). This is accomplished by enabling the bit 0-5 coefficient result (Xi) transmitters on modules K01 and K02. Term "C" becomes a "one" when the following conditions exist:

(Pop. Count)(Transmit)(t30)

Notice that pins 16 and 26 on K02 are tied to +1.2 volts ("zero") to disable the transmission of "ones" in bits 2^6 and 2^7. On the remaining transmitter modules, the term "C" input, pin 17,



POPULATION COUNT MODE BIT

Figure 7.6-5

is left open forcing "zero" out of C and, consequently, on the data trunk. Hence, bits 6-59 of the population count result are always "zeros".

3) Disable the setting of the Xj or Xk indefinite or infinite flip-flops (via G17, pin 20). These flip-flops are located on Module A38, test points 1 and 2. Test point 1 is set when Xj or Xk bits 48-59 have the octal configuration, 1777 or 6000 (± indefinite operand) and pin 5 is a "one". Test point 2 is set when Xj or Xk bits 48-59 have the octal configuration, 3777 or 4000 (± infinity) and pin 7 is a "one". Both pins 5 and 7 translate as:

$$\overline{(\text{Pop. Count})} + (\text{Divide}) + (\text{Multiply 1}) + (\text{Multiply 2})$$

and are therefore "zeros" when in the Population Count mode. In this case the flip-flops will not be set, regardless of the bit 48-59 configurations, and no divide error signal can be generated.

4) Disable the Error Mode Test logic via G17, pin 24. This is accomplished to prevent a Divide error signal from being generated in the event that the combination of the Xj or Xk exponent generates a non-standard operand. In essence, the Error Test circuitry is disabled when the "Population Count" mode bit is present.

7.6.4 QUOTIENT TIMING SEQUENCE

This discussion deals primarily with the quotient timing sequence,as opposed to exponent timing which is discussed in Section 7.6.8.

The quotient timing sequence is initiated upon receipt of the "Go Divide" signal from the scoreboard. A timing chain is located on modules D21 through D26 (C. E. Diagrams, sheets 171 and 172) and is composed of 36 flip-flops arranged in a chain. Each flip-flop is clear/set 75 nanoseconds after the preceding stage in the chain (similar to the PPU barrel timing concept). For example, the first flip-flop is set by a t15, the second with t90, the next with t65, the next with t40, etc. Thus, approximately 2700 nanoseconds elapse from setting the first flip-flop (D21, TP1) to setting the last (D26, TP6). Each flip-flop is set for 1 minor cycle. The outputs of these flip-flops, singly and in combination are used to sequence the manipulations of the Xj and Xk coefficients required to form the coefficient and exponent of the quotient.

An explanation of each term on the quotient timing chart (Figure 7.6-6) follows. In conjunction with the timing chart, the C. E. Diagrams and Chassis 2 wire tabs should be utilized to further clarify the discussion of quotient generation.

1) SCOREBOARD ISSUE - This is the time reference for the timing chart - the scoreboard issue of the Divide opcode.
2) GO DIVIDE - Assuming that no second order conflicts occur, the "Go Divide" signal will be received on Chassis 2 approximately 175 nanoseconds after Scoreboard issue. This signal initiates the divide timing chain.

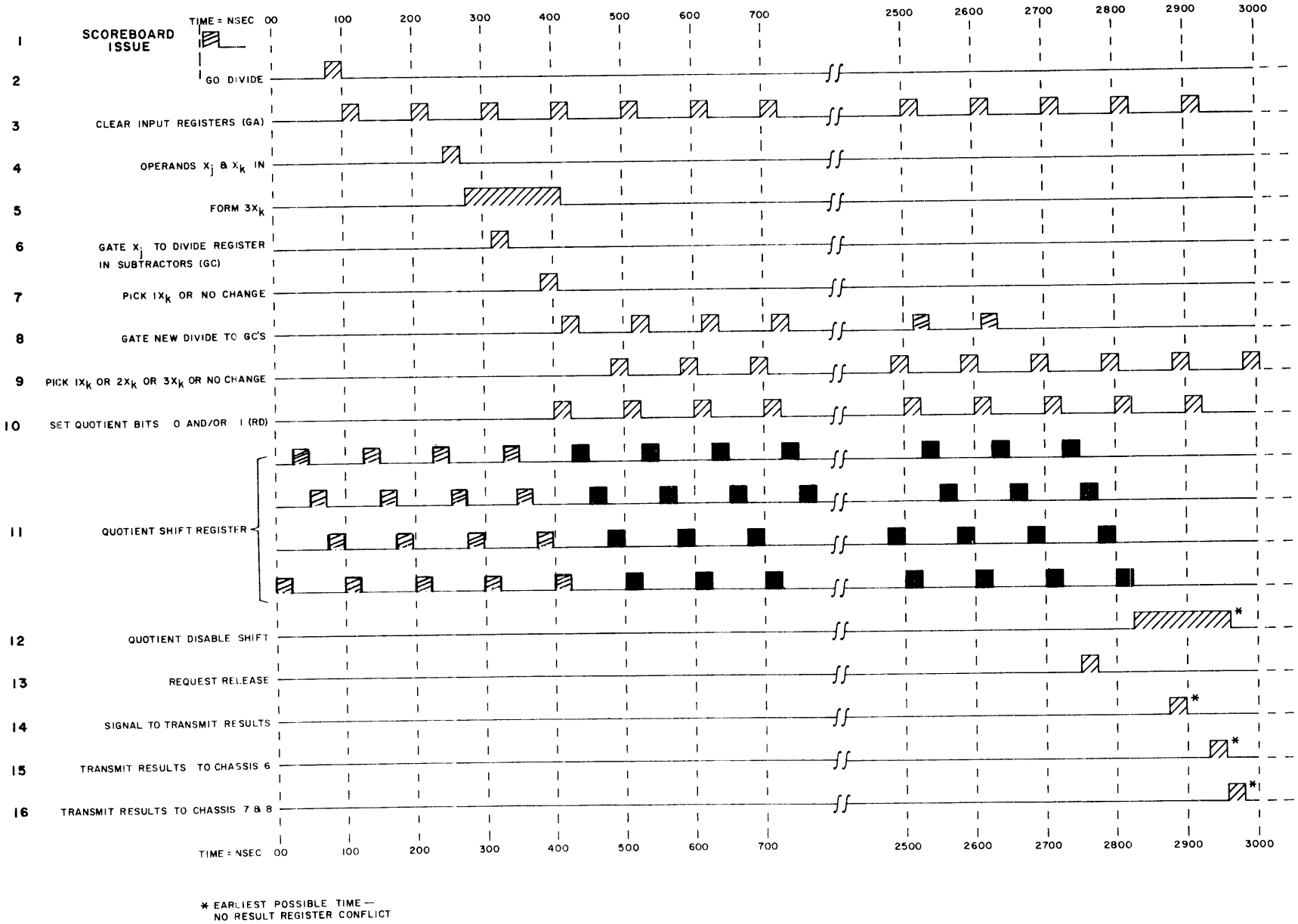

TIME = NSEC 00 100 200 300 400 500 600 700 2500 2600 2700 2800 2900 3000
1 SCOREBOARD ISSUE
2 GO DIVIDE
3 CLEAR INPUT REGISTERS (GA)
4 OPERANDS X_j & X_k IN
5 FORM $3X_k$
6 GATE X_j TO DIVIDE REGISTER IN SUBTRACTORS (GC)
7 PICK $1X_k$ OR NO CHANGE
8 GATE NEW DIVIDE TO GC'S
9 PICK $1X_k$ OR $2X_k$ OR $3X_k$ OR NO CHANGE
10 SET QUOTIENT BITS 0 AND/OR 1 (RD)
11 QUOTIENT SHIFT REGISTER
12 QUOTIENT DISABLE SHIFT
13 REQUEST RELEASE
14 SIGNAL TO TRANSMIT RESULTS
15 TRANSMIT RESULTS TO CHASSIS 6
16 TRANSMIT RESULTS TO CHASSIS 7 & 8
TIME = NSEC 00 100 200 300 400 500 600 700 2500 2600 2700 2800 2900 3000
* EARLIEST POSSIBLE TIME — NO RESULT REGISTER CONFLICT

Figure 7.6-6

3) CLEAR INPUT REGISTERS (GA) - The input registers are cleared every minor cycle as shown, at approximately t 00.

4) OPERANDS Xj & Xk IN - The Xj and Xk operands will be received on chassis 2 (from chassis 7 & 8) at approximately t 250.

5) FORM 3 Xk - No logic gate exists to enable the formation of 3 Xk. This term is shown to indicate the time allowed for the generation of 3 Xk-the time from entering the operands until the first iteration begins.

6) GATE Xj to DIVIDE REGISTER - This signal occurs just prior to the first iteration to enable the original dividend, Xj, into the dividend registers. Xj is gated via the GF modules which are also used to gate the previous partial dividend left shifted two places, when the "Pick No Change" gate is a "one". (C. E. Diagrams, sheet 175).

7) PICK 1Xk or NO CHANGE - This gate occurs along with the Pick 2Xk or 3Xk gate on all iterations except the first. On the first iteration, the Pick 2Xk or 3Xk gate is disabled and only Pick 1Xk or No Change occurs. This happens because if both initial operands have been normalized (as they should be) the first (upper) 2 bits of the quotient can only be 01 or $00_{(2)}$. Enabling the Pick 2Xk or 3Xk on the first iteration would only insure a meaningless result. This logic can be seen in the C. E. Diagrams on sheet 183.

8) GATE NEW DIVIDEND TO GC's - This signal occurs 24 times during the divide step. Each time, the new partial dividend is gated from the output of one of the three subtractors back to the

three dividend registers of the subtractors. If the Xj-Xk subtractor had an End Around Borrow, the previous dividend is gated back to the dividend registers, left shifted two places, by the "Pick No Change" gate. If the Xj-Xk subtractor had no EAB, but the Xj-2Xk subtractor did, the output of the Xj-Xk subtractor is used as the next partial dividend. The following table shows all possibilities and includes the quotient bits generated for each case.

TABLE 7.6-1

CONDITION	PICK	QUOTIENT
(3Xk=$\overline{EAB}$)	3Xk	11
(3Xk=EAB)(2Xk=$\overline{EAB}$)	2Xk	10
(2Xk=EAB)(Xk=$\overline{EAB}$)	Xk	01
(Xk=EAB)	NO CHANGE	00

9) PICK 1Xk OR 2Xk OR 3Xk OR NO CHANGE - This gate occurs every minor cycle after time 400 and selects one of four values (Picks) to be gated to the dividend registers for the next iteration (See table 7.6-1 and term #8). Although the selection continues to occur after the last iteration, the selection will not be useful because the left shift of the quotient shift register is disabled after the 24th iteration (See term #11).

10) SET QUOTIENT BITS 2^0 and/or 2^1 (RD) - The setting of quotient bit 2^0 is enabled on each t90 after t350 of the divide timing chain while setting 2^1 is enabled on each t90 after t410. This occurs since the setting of 2^1 should not be possible on the first iteration if the original operands are normalized (See term #7). Although the setting of the quotient bits is enabled even after the last iteration, the final quotient is not affected

because the left shifting of the quotient shift register is disabled after the 24th iteration. (See term #11).

11) QUOTIENT SHIFT REGISTER - A two place left-shift register is used to assemble the 2-bit quotients into the final 50-bit quotient. The quotients are selected at 100 nanosecond intervals according to the trial subtractions, and are inserted into stages 0 and 1 of the left shift register (See Figure n7.6-7 and the C. E. Diagrams, sheets 183 and 184). Since

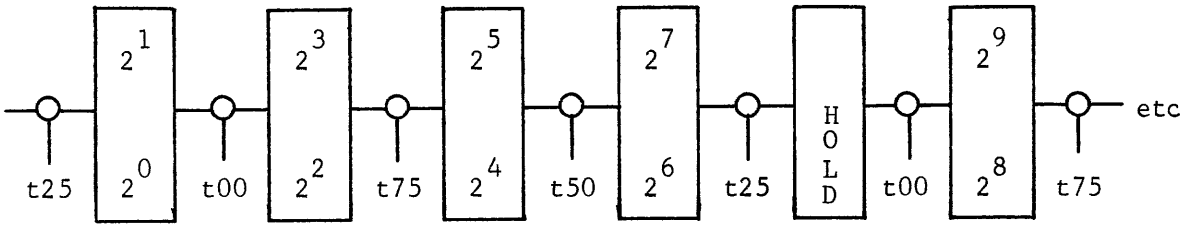


Figure 7.6 - 7

this register performs the left shifts at 75 nanosecond intervals, two "Holding" stages are required after each 4 shifts to allow the picking of quotients to "catch up" with the shifting of the register.

This term on the timing chart shows, in solid black, the shifts which are meaningful to the generation of the final quotient. The other shifts occur prior to the generation of any quotient bits and are therefore superfluous.

12) QUOTIENT DISABLE SHIFT - The left shifting of the quotient shift register is disabled after the 24th and final iteration. This is done by a flip-flop (C. E Diagrams, sheet 184, E10, TP1) which is set at divide time 2775. The flip-flop is cleared upon receipt of the "Transmit" signal from the score board. The earliest time possible for receipt of "Transmit" is about 2875.

13) REQUEST RELEASE - The Request Release is sent to the scoreboard at about t2750 of the Divide sequence. It resolves any third order conflicts which may exist.

14) SIGNAL TO TRANSMIT RESULTS - Assuming that no third order conflicts exist, the signal to transmit the result will be received on chassis 2 at about t2875.

15) TRANSMIT RESULTS - Assuming that no third order conflicts exist,
&
16) bits 0-47 of the result will be transmitted to chassis 6 from the RM modules at t2930. From there they are gated to chasses 7 & 8 about 50 nanoseconds later. Bits 48-59 of the result are transmitted directly to chassis 7 & 8 (from RM modules) at t2960.

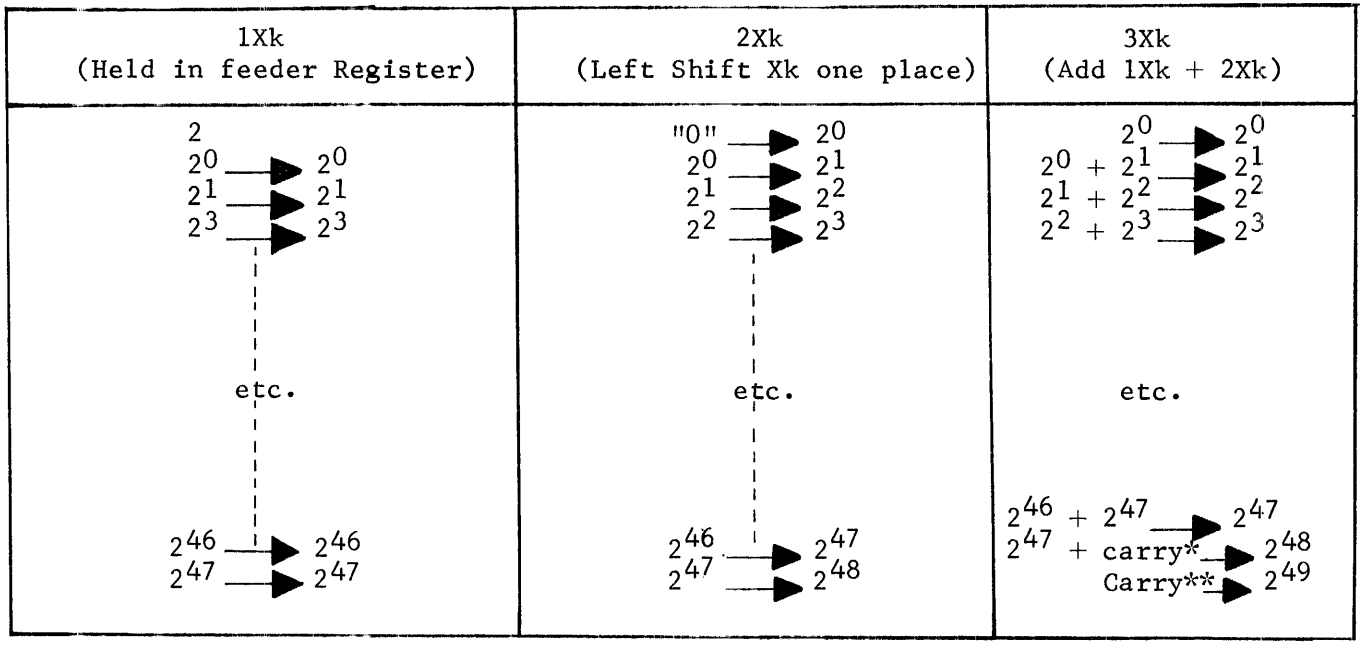
7.6.5 1, 2, 3 TIMES DIVISOR (Xk)

The function of the logic discussed in this section is to 1) form the values 1, 2 and 3 times the divisor (Xk) and 2) distribute each value to its respective adder.

At the beginning of the divide operation, the divisor (Xk) is placed in its holding register located on MA modules. It remains there, unattended, throughout the divide step. This register feeds a static network which forms 1, 2 and 3 times Xk. Since 1Xk exists in the holding register, no aditional logic is required for its formation. 2Xk is formed by left shifting the divisor 1 bit position, thereby multiplying by two. 3Xk is formed with a special full adder which adds 1Xk and 2Xk. Due to the propagation of carries to high order bits, the value, 3Xk, may be 50 bits in length. (See Figure 7.6-9 .)

Figure 7.6-8 should be used in the following logic analysis of the 1, 2 and 3Xk circuitry. The 3Xk adder is not shown completely (the carry determination and propagation logic is omitted), so reference to the C. E. Diagrams (Sheet 177) and the Chassis 2 wire tabs should also be made.

Since the value 1Xk is already formed in the holding (MA modules) register it is simply gated directly to the GC modules of the Xj - Xk adder. For example bit 2^0 is fed from R11, pin 12, via Q19, pins 1 and 13, to M28, pin 6, which is an input to adder stage 2^0. On Q19, gating term "A" is always a "one" since the input pin 6 is always a "zero". The remaining bits of 1Xk are unconditionally gated to stages 1-47 of the adder in the same manner.



* Possible carry from stage 47
** Possible carry from stage 48

REQUIREMENTS FOR FORMATION OF 1, 2, AND 3Xk

Figure 7.6-9

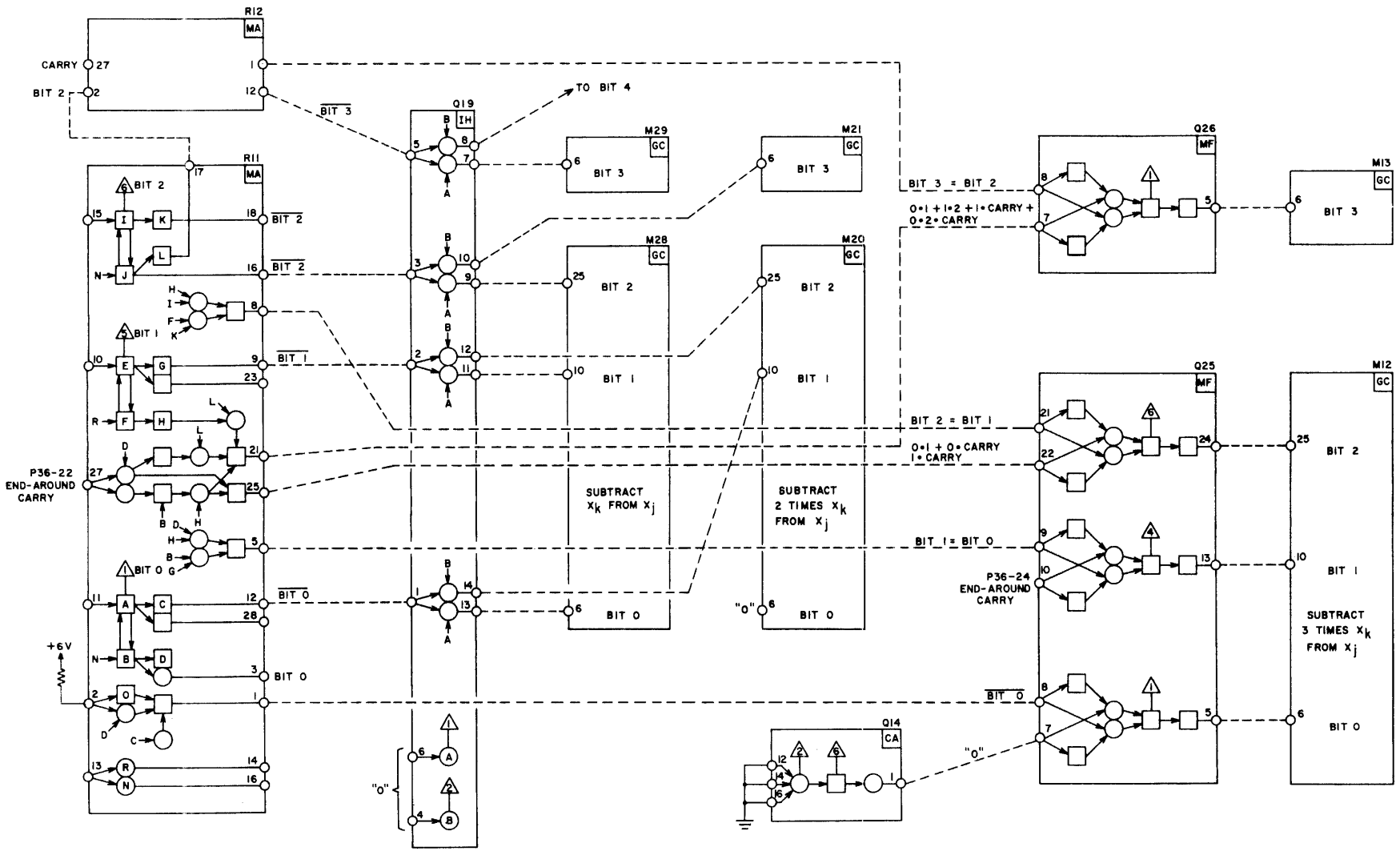

R12
MA
CARRY
BIT 2
TO BIT 4
BIT 3
Q19
IH
M29
GC
M21
GC
R11
MA
BIT 2
BIT 2
BIT 2
M28
GC
M20
GC
BIT 1
BIT 1
BIT 1
SUBTRACT
X_k FROM X_j
SUBTRACT
2 TIMES X_k
FROM X_j
P36-22
END-AROUND
CARRY
BIT 0
BIT 0
BIT 0
"0"
+6V
"0"
Q14
CA
Q26
MF
M13
GC
BIT 3 = BIT 2
0•1 + 1•2 + 1• CARRY +
0•2• CARRY
BIT 3
Q25
MF
M12
GC
BIT 2 = BIT 1
0•1 + 0• CARRY
1• CARRY
BIT 2
BIT 1 = BIT 0
P36-24
END-AROUND
CARRY
BIT 1
SUBTRACT
3 TIMES X_k
FROM X_j
BIT 0
BIT 0
"0"

Figure 7.6-8

The value 2Xk is gated in a manner similar to 1Xk with the exception that each bit position is left shifted one place. Bit 2^0 of Xk is therefore fed to stage 2^1 of the adder, bit 2^1 of Xk to stage 2^2, etc. The stage 2^0 input is a constant "zero" (M20, pin 6) since a "one" value is never possible when left shifting Xk.

3Xk ADDER:

The 3Xk Adder is a full adder which forms the sum of Xk and 2Xk. Since 2Xk is formed by left shifting Xk one bit position, only one feeder register is required. Each bit of the feeder register (MA modules) has two outputs to the adder logic. One output is to the corresponding stage of the adder; the second is to the next significant bit position, as follows:

Xk bits			47	46	45	----------	3	2	1	0
2Xk bits		47	46	45	44	----------	2	1	0	
3Xk bits	49	48	47	46	45	----------	3	2	1	0

As can be seen, the result may be 50 bits in length because of the possible carry into bit 2^{49}. Note also, that bit 2^0 of the result will always be the same as bit 2^0 of Xk since nothing is added to that bit position.

A true addition is performed by the adder (as opposed to subtracting to add, complementing operands, etc.). As a result, the conditions generate, satisfy, enable and pass are defined as follows:

	Generate	Satisfy	Enable
Xk	1	0	1 or 0
2Xk	1	0	0 or 1

The condition $\overline{\text{Satisfy}}$ is also referred to as a Pass.

The addition is performed in three basic logical steps:

1) Determine whether or not Equivalence exists between the two source operands.

   i.e. EQUIVALENCE ⇒ $\begin{matrix}1\\1\end{matrix}$ or $\begin{matrix}0\\0\end{matrix}$

2) Determine into which stages carries are entered. A carry will enter a stage if the previous stage is a generate or if some other less significant stage is a generate and none of the more significant stages are satisfied.

3) Toggle the results of the equivalence checks for all stages that do not have a carry in.

The following example illustrates the preceding steps. The original value of Xk is assume to be 0 $01234567_{(8)}$.

NOTE: 1) S,E, and G refer to Satisfy, Enable, and Generate respectively.

2) Asterisks indicate those stages with a carry in.

				*	**	**	***	***	*	
	SS	S	SEE	EES	EGG	ESE	EEG	GEE	GGE	
Xk =	0	0	001	010	011	100	101	110	111	(2)
2Xk =	00	0	010	100	111	001	011	101	110	
STEP 1 (Equivalence) =	11	1	100	001	011	010	001	100	110	
STEP 2 (Carry In) =	00	0	000	001	110	011	111	111	100	
STEP 3 =	00	0	011	111	010	110	001	100	101	
IN OCTAL =	0	0	3	7	2	6	1	4	5	

Using octal arithmetic, the same result is obtained:

Xk = 0———01234567
2Xk = 0———02471356
3Xk = 0———03726145

The following formulas which define the sum as a "one" or a "zero" can be derived from the above procedure:

Sum = "1" if Equivalence · Carry + $\overline{\text{Equivalence}}$ · $\overline{\text{Carry}}$

Sum = "0" if $\overline{\text{Equivalence}}$ · Carry + Equivalence · $\overline{\text{Carry}}$

Several module types are used in the 3Xk adder. Their functions are generally described as follows:

MA - Contain feeder register, make initial equivalence check, and propagate carries within a 3-bit group (bits 0-2, 3-5, 6-8, etc.).

ME - Determine the pass and carry-out conditions for 3-bit groups.

MH - Determine carry-outs for six sections (bits 0-9, 10-18, 19-27, 28-36, 37-45, 46-47). Also, final carries into stages are determined for some bits.

MI - Sum up the group and section Pass conditions.

MJ & MH - Make carry propagation checks for carries into 3-bit groups.

MF - Perform final equivalence and carry summation and produce the final sum.

A portion of the 3Xk adder logic is shown in Figure 7.6- ,but the carry and pass summation logic is left out. The Chassis 2 wire tabs should therefore be used to understand the adder logic completely.

The equivalence check, which is made on the MA modules, is a simple matter of checking the inputs to each adder stage for a 0/0 or 1/1 configuration. The logic is shown for stages 1, 2 and 3 in Figure 7.6-8 . (The check need not be made for bit 2^0 since 2^0 of the result will always be the same as bit 2^0 of Xk.) Pin 5, for example, translates as:

$$2^0 \cdot 2^1 + \overline{2^0} \cdot \overline{2^1}$$

or, Equivalence in stage 2^1

Pin 8 translates as:

$$2^1 \cdot 2^2 + \overline{2^1} \cdot \overline{2^2}$$

or, Equivalence in stage 2^2

This check occurs for all bit positions on the MA modules, which are located on R11 through R26 (bits 0-47 of Xk). The result of the equivalence check are sent to the MF modules and are ANDed with the carry-in conditions to generate the final product.

The final sum is generated on the MF modules (see Figure 7.6-8) with an equivalence circuit. For example, 2Q25, TP4 is the equivalence circuit for stage 2^1 of the adder. If the two inputs (pins 10 and 9 which are the stage equivalence and carry-in conditions respectively) are both ones and both zeros, the output on pin 13 will be a "one". The outputs of the test points (i.e. #4) are actually the false value of the sum, 3Xk and are fed to the adder inputs (GC modules).

Since a relatively comprehensive understanding of other 6600 adders is assumed, further discussion of the 3Xk Adder is felt to be somewhat wasteful. At this point, the concept of the adder should be quite clear. If further analysis is desired, this concept should guide research of the Chassis 2 wire tabs.

7.6.6 SUBTRACT Xk from Xj

Since all three coefficient subtractors operate similarly, only one , Xj - Xk, is discussed. The subtractors form the difference logically, according to the following rules:

	GENERATE	SATISFY	ENABLE	
Xj	0	1	1	0
Xk	1	0	1	0

The final summation logic generates a difference of "one" if the following conditions are met:

$$(\text{EQUIVALENCE})(\text{BORROW IN}) + (\overline{\text{EQUIVALENCE}})(\overline{\text{BORROW IN}})$$

EXAMPLE:

In Octal:	Xj	= 67261
	Xk	= 25652
	DIFFERENCE	= 41407

Machine Method:

Stage Definition		SEE	ESE	GEE	ESG	EGS
Xj	=	110	111	010	110	001
Xk	=	010	101	110	101	010
Equivalence	=	011	101	011	100	100
Borrow In	=	000	011	000	011	100
$(EQ \cdot B) + (\overline{EQ} \cdot \overline{B})$	=	100	001	100	000	111
In octal	=	4	1	4	0	7

Figure 7.6-10 shows the lower three stages of the Xj - Xk subtractor. The GC modules make equivalence, generate, and satisfy checks according to the above rules. For stage 2°:

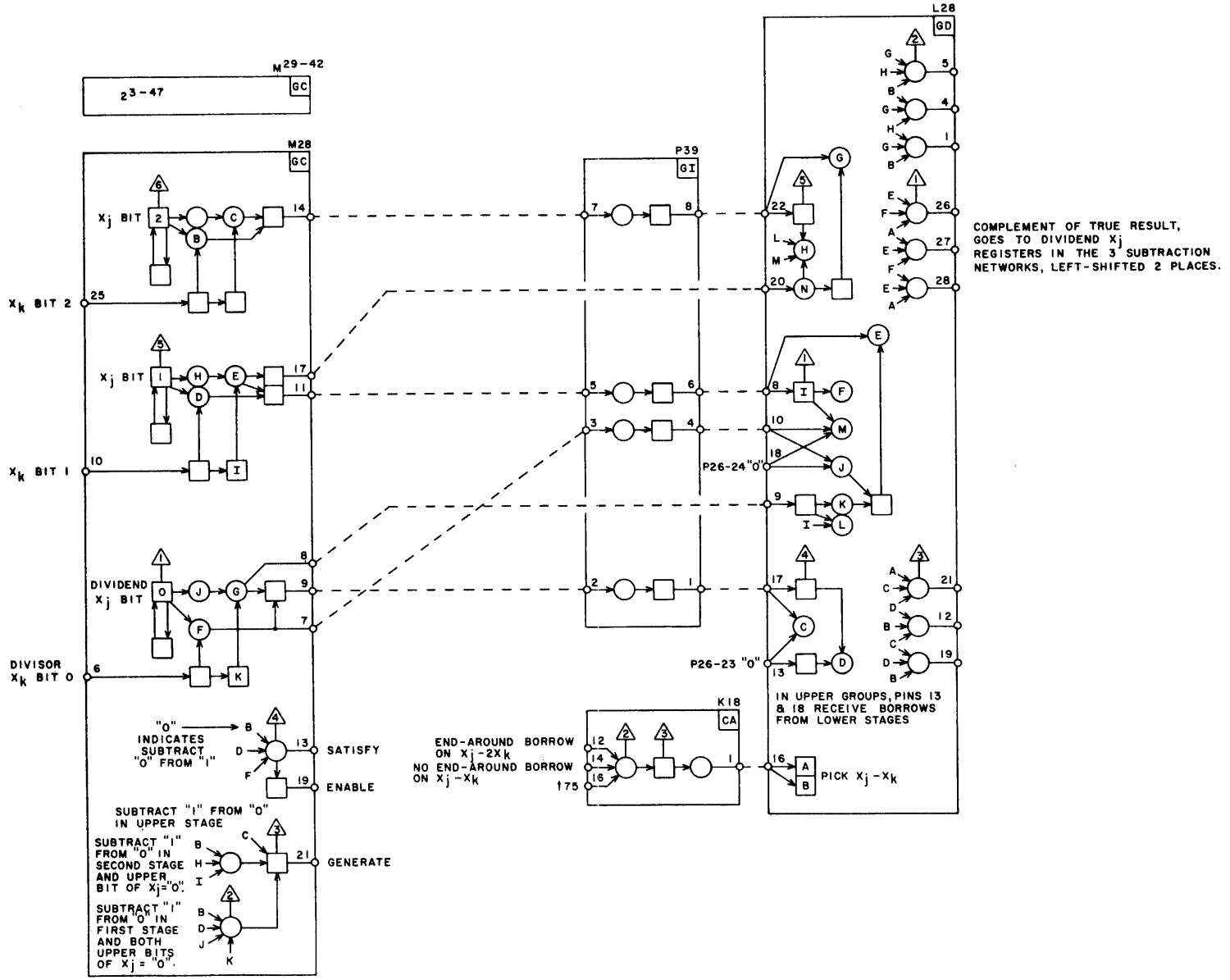

M29-42
2^{3-47}
GC
M28
GC
X_j BIT
X_k BIT 2
X_j BIT
X_k BIT 1
DIVIDEND X_j BIT
DIVISOR X_k BIT 0
"0" INDICATES SUBTRACT "0" FROM "1"
SATISFY
ENABLE
SUBTRACT "1" FROM "0" IN UPPER STAGE
SUBTRACT "1" FROM "0" IN SECOND STAGE AND UPPER BIT OF X_j="0".
SUBTRACT "1" FROM "0" IN FIRST STAGE AND BOTH UPPER BITS OF X_j = "0".
GENERATE
P39
GI
P26-24 "0"
P26-23 "0"
K18
CA
END-AROUND BORROW ON X_j-2X_k
NO END-AROUND BORROW ON X_j-X_k
PICK X_j-X_k
L28
GD
COMPLEMENT OF TRUE RESULT, GOES TO DIVIDEND X_j REGISTERS IN THE 3 SUBTRACTION NETWORKS, LEFT-SHIFTED 2 PLACES.
IN UPPER GROUPS, PINS 13 & 18 RECEIVE BORROWS FROM LOWER STAGES

Figure 7.6-10

pin 7 $\Longrightarrow$ $\overline{\overline{Xj} \cdot \overline{Xk}}$ $\Longrightarrow$ $\overline{Satisfy}$

pin 8 $\Longrightarrow$ $\overline{Equivalence}$

pin 9 $\Longrightarrow$ $\overline{\overline{Xj} \cdot Xk}$ $\Longrightarrow$ $\overline{Generate}$

These are sent (via P39) to L28, the final summation network. Keep in mind that an end around borrow indicates that the Xk multiple is greater than Xj in which case the difference is not used. Only if there is no EAB, will the difference be useful. Thus, bit 2^0 will never have a borrow input. Thus, the formula for a "1" in 2^0 of the result is simply $\overline{Equivalence}$ (L28, term D). The output for bit zero is taken from pins 21, 12 & 19. This is the complement of the true out-put and therefore can directly set the feeder register flip-flops. It is gated only if the "Pick Xj - Xk" gate is a "one". In the same manner, the output for 2^1 is taken from pins 5, 4 & 1 and for bit 2^2 from pins 26, 28 & 27 - again, in complement form.

Stage 2^1, of course, will have a borrow input if bit 2^0 generates (L28,9). L28,10 indicates $\overline{Satisfy}$ in bit 2^0, but this input will not be used since an EAB can never occur. L28,8 indicates $\overline{Equivalence}$ in stage 2^1. Translation of terms "E" and "F" yield the following formulas.

L28, "E" $\Longrightarrow$ $EQ + \overline{B}$

L28, "F" $\Longrightarrow$ $\overline{EQ} + B \Longrightarrow$ (term J = constant "1")

These terms are ANDed at the 2^1 output circuits (pins 5, 4 & 1) to yield the following translation for a difference of "1" (zero output):

$$E \cdot F \Longrightarrow (EQ + \overline{B})(\overline{EQ} + B) \Longrightarrow EQ \cdot B + \overline{EQ} \cdot \overline{B}$$

This corresponds to the initial machine method example. The remaining bit positions are logically similar. Proof of their operation is therefore left to the reader.

7.6.7 QUOTIENT OUTPUT NETWORK

After completing 24 divide iterations, the coefficient of the final quotient is held in the quotient shift register until receipt of the "Transmit" signal from the score board. When the "transmit" is received, either the true or complemented value of the quotient will be sent on the data trunk. If bit 2^{48} of the quotient shift register is set, the quotient must be right shifted one place before being transmitted. Also, if an error condition exists (i.e. Underflow, Overflow, or Indefinite Result) the coefficient must be made all zeros. The function of the output network is to select and transmit the proper quotient.

The right shift logic is shown in Figure 7.6-12. If bit 2^{48} of the quotient is set (E13, TP6) and the quotient is ready (E10, TP1), terms B and D on J 16 will be "ones". This enables bit 1 of the result to the bit 0 transmitter, bit 2 to bit 1, bit 3 to bit 2, etc., and in this manner performs the right shift one place. If 2^{48} is a zero, terms A and C on J 16 are ones (B and D are "zeros") to enable each bit to its respective transmitter and thus disable the right shift one place.

Note that terms A and B on the RM Modules translate as True and Complement respectively. The logic which determines whether the true or false value will be selected is shown on sheet 186 of the C.E. Diagrams. Since the logic requires quite a detailed analysis, only the concept will be presented here. The actual proof of the logic is, in most cases, left to the student.

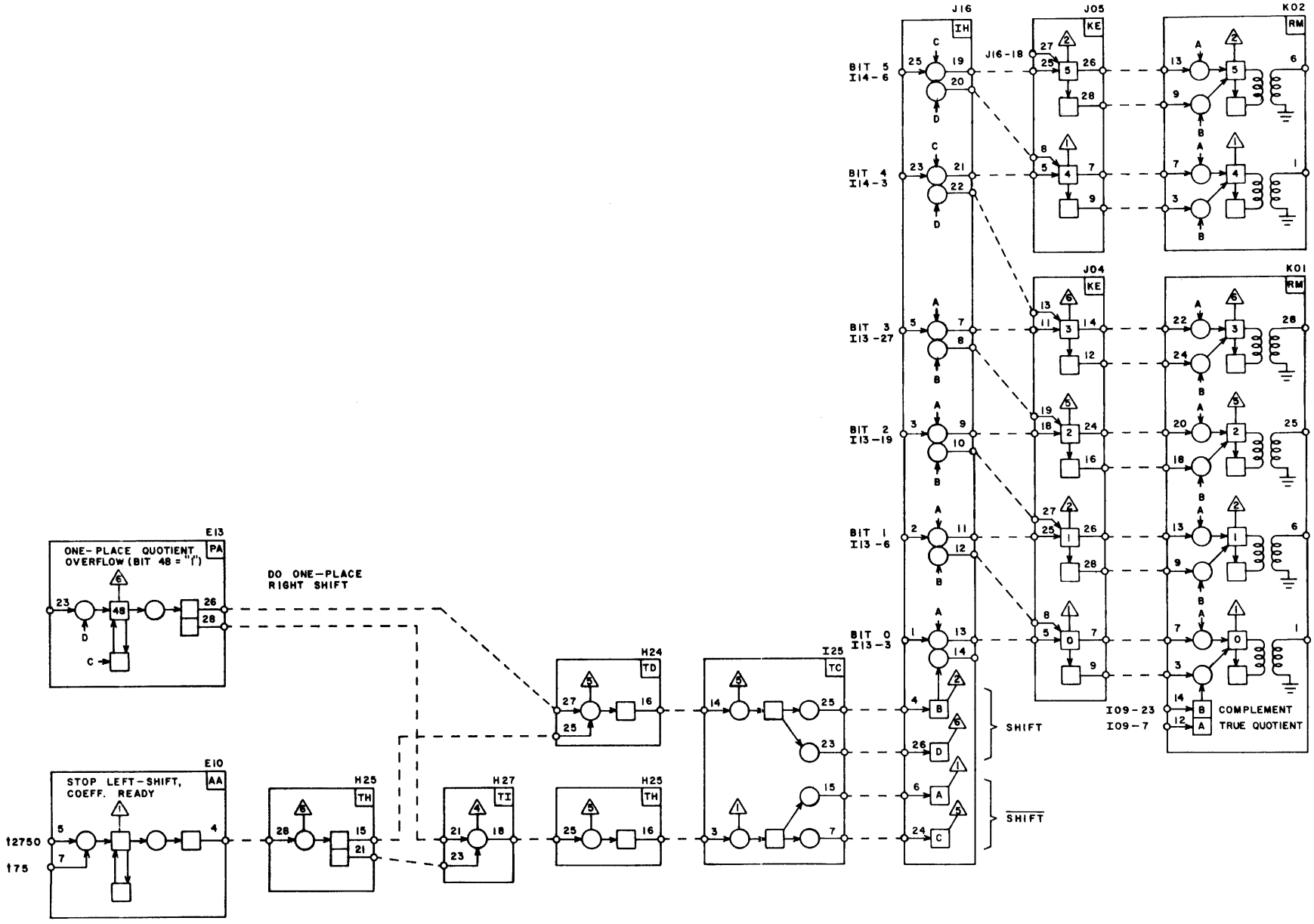

E13
ONE-PLACE QUOTIENT OVERFLOW (BIT 48 = "1")
PA
DO ONE-PLACE RIGHT SHIFT
E10
STOP LEFT-SHIFT, COEFF. READY
AA
12750
175
H25
TH
H27
TI
H24
TD
I25
TC
J16
IH
J05
KE
J04
KE
K02
RM
K01
RM
BIT 5 I14-6
BIT 4 I14-3
BIT 3 I13-27
BIT 2 I13-19
BIT 1 I13-6
BIT 0 I13-3
J16-18
SHIFT
$\overline{\text{SHIFT}}$
I09-23
I09-7
COMPLEMENT TRUE QUOTIENT

Figure 7.6-12

When the original coefficient signs are alike (i.e. 1/1 or 0/0) the quotient should have a positive sign. If the signs were unlike (i.e. 0/1 or 1/0) the quotient should have a negative sign. Test Point 3 on I07 will output a "zero" to enable selection of the false coefficient if the signs were unlike AND the quotient is NOT infinite, indefinite or zero. (The existance of these conditions is determined by checking the original operands and the exponent of the result.) Test Point 1 will output a "zero", to enable selection of the true coefficient, if the signs are like AND the quotient is NOT infinite, indefinite or zero. If one of the error conditions does exist, both test points will output "ones" and thereby cause terms A and B on the RM modules (figure 7.6-12) to be zeros. This disables the coefficient transmitters and causes "zeros" to be sent in bits 0-47 of the result.

7.6.8 EXPONENT ADDERS

Three adders, shown in Figure 7.7-13, are required for the formation of the final exponent:

The first forms the algebraic difference of the two exponents, xj -xk.

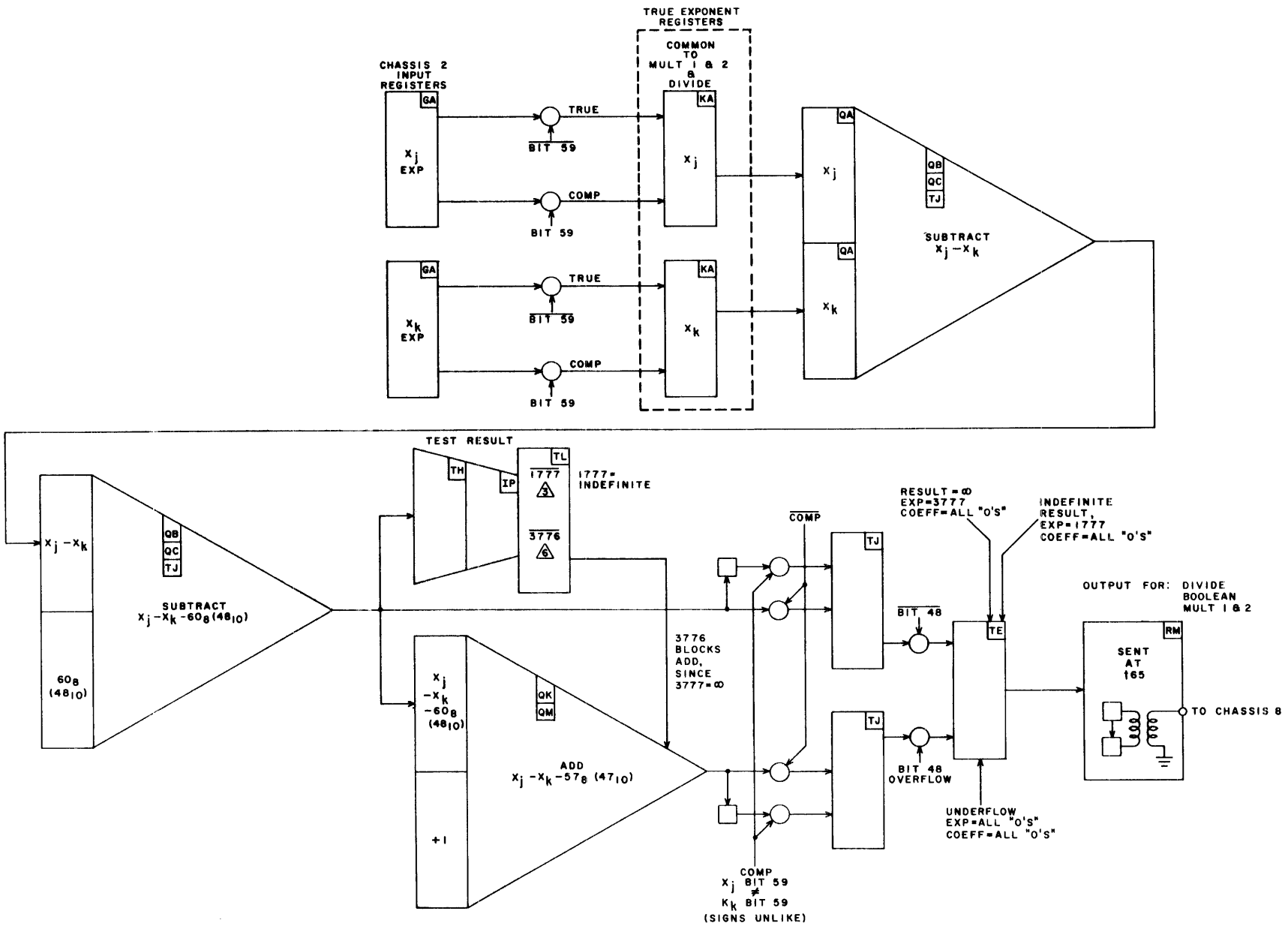
The second subtracts $60_{(8)}$ from the result of the first. This value is used as the final exponent if right shifting one place to normalize is not performed.

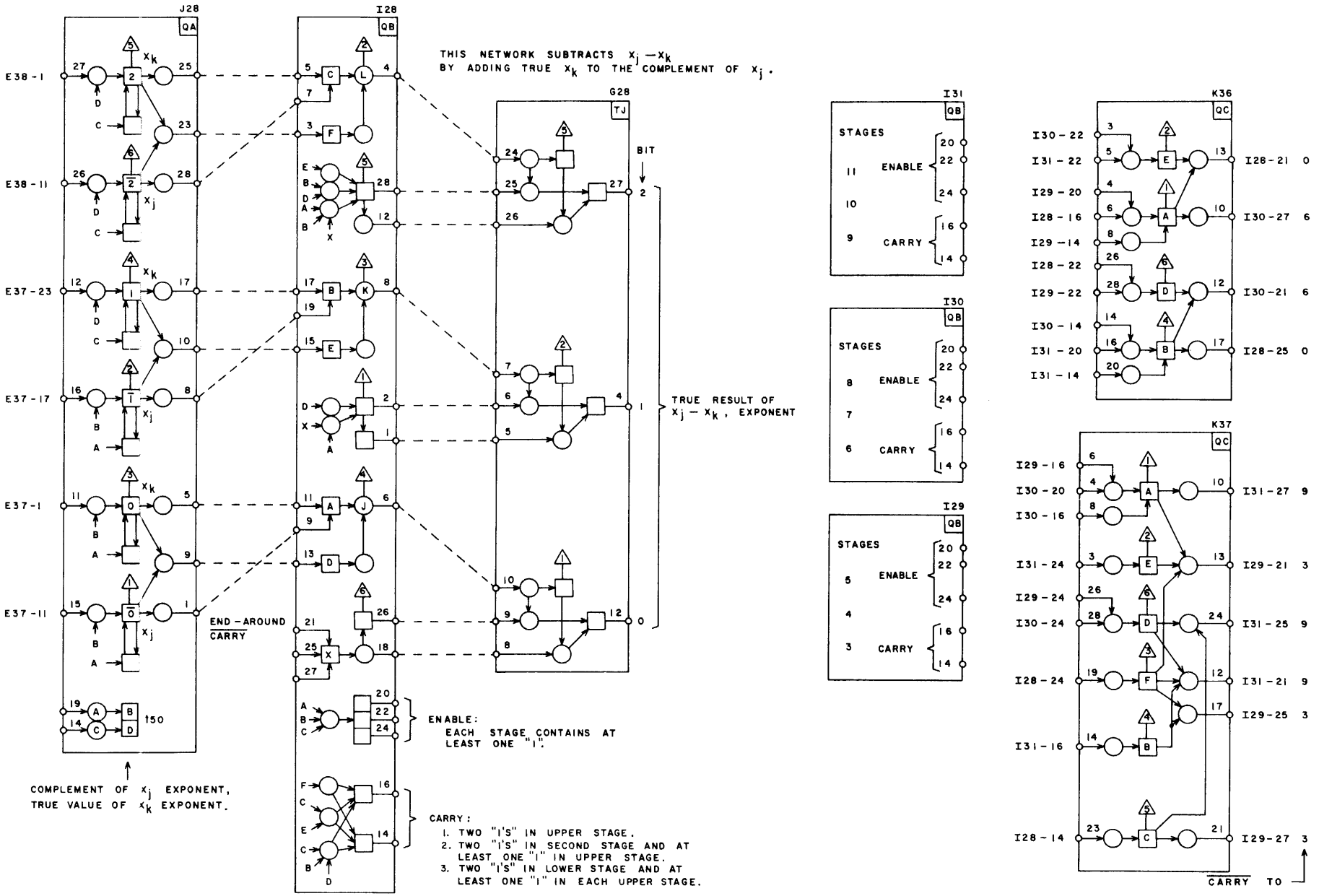
The third adds 1 to the result of the second adder. This value is used if a right shift one place is required to normalize the quotient.

Xj-Xk SUBTRACTOR:

Refer to Figure 7.6-14 (Xj-Xk adder logic) during the following discussion. The feeder registers are located on QA modules and hold the complement of the Xj exponent and the true value of the Xk exponent. Recall that this adder, using QA, QB, QC, and TJ modules (like the Peripheral Process or A and Q Adders and the Multiply Xj +Xk Adder) normally forms a sum by adding the complements of numbers and then complementing the result. Since a difference of the exponents is required, and is accomplished by adding the complement of Xk to Xj, the Xk feeder contains a true value and the Xj feeder a false value. For example, in forming A - B:

TRUE ADDER	ADDING COMPLEMENTS
A = 0125	$\overline{A}$ = 7652
$\overline{B}$ = 7723	B = 0054
0050	DIFF= 7706
1	$\overline{DIFF}$= 0051
DIFF= 0051	


TRUE EXPONENT REGISTERS
COMMON TO MULT 1 & 2 & DIVIDE
CHASSIS 2 INPUT REGISTERS
GA
x_j EXP
TRUE
BIT 59
COMP
BIT 59
KA
x_j
QA
x_j
QB
QC
TJ
SUBTRACT $x_j - x_k$
GA
x_k EXP
TRUE
BIT 59
COMP
BIT 59
KA
x_k
QA
x_k
TEST RESULT
TH
IP
TL
1777
3
3776
6
1777= INDEFINITE
$x_j - x_k$
QB
QC
TJ
SUBTRACT $x_j - x_k - 60_8 (48_{10})$
60_8 (48_{10})
$x_j - x_k - 60_8$ (48_{10})
QK
QM
ADD $x_j - x_k - 57_8$ (47_{10})
+1
3776 BLOCKS ADD, SINCE 3777=∞
COMP
TJ
TJ
RESULT = ∞ EXP=3777 COEFF=ALL "0'S"
INDEFINITE RESULT, EXP=1777 COEFF=ALL "0'S"
BIT 48
BIT 48 OVERFLOW
TE
UNDERFLOW EXP=ALL "0'S" COEFF=ALL "0'S"
OUTPUT FOR: DIVIDE BOOLEAN MULT 1 & 2
RM
SENT AT 165
TO CHASSIS 8
COMP x_j BIT 59 ≠ K_k BIT 59 (SIGNS UNLIKE)


J28
QA
I28
QB
G28
TJ
THIS NETWORK SUBTRACTS $x_j - x_k$ BY ADDING TRUE x_k TO THE COMPLEMENT OF x_j.
E38-1
E38-11
E37-23
E37-17
E37-1
E37-11
x_k
x_j
BIT
2
1
0
TRUE RESULT OF $x_j - x_k$, EXPONENT
END-AROUND CARRY
150
COMPLEMENT OF x_j EXPONENT, TRUE VALUE OF x_k EXPONENT.
ENABLE:
EACH STAGE CONTAINS AT LEAST ONE "1".
CARRY:
1. TWO "1'S" IN UPPER STAGE.
2. TWO "1'S" IN SECOND STAGE AND AT LEAST ONE "1" IN UPPER STAGE.
3. TWO "1'S" IN LOWER STAGE AND AT LEAST ONE "1" IN EACH UPPER STAGE.
I31
QB
STAGES
11
10
9
ENABLE
CARRY
20
22
24
16
14
I30
QB
STAGES
8
7
6
ENABLE
CARRY
I29
QB
STAGES
5
4
3
ENABLE
CARRY
K36
QC
I30-22
I31-22
I29-20
I28-16
I29-14
I28-22
I29-22
I30-14
I31-20
I31-14
I28-21 0
I30-27 6
I30-21 6
I28-25 0
K37
QC
I29-16
I30-20
I30-16
I31-24
I29-24
I30-24
I28-24
I31-16
I28-14
I31-27 9
I29-21 3
I31-25 9
I31-21 9
I29-25 3
I29-27 3
CARRY TO

Either method yields the correct answer (i.e. 125 - 54 = 51).

In the true adder,

a) $\text{EQUIVALENCE} \cdot \text{CARRY IN} + \overline{\text{EQUIVALENCE}} \cdot \overline{\text{CARRY IN}} = 1$

b) $\overline{\text{EQUIVALENCE}} \cdot \text{CARRY IN} + \text{EQUIVALENCE} \cdot \overline{\text{CARRY IN}} = 0$

In order to add complements and obtain a true sum (which is, in this case, a difference since Xk is complemented), it is necessary to complement the result of the addition. In the Xj - Xk adder this is accomplished by reversing the meaning of a carry. A carry will be generated from stage "X" if both the feeder registers contain "ones" or if a carry enters stage X from X-1 and stage X is not a satisfy condition. But, since the feeders contain the complemented values of Xj and Xk, a generate condition (1/1) indicates a satisfy as far as the true values are concerned. In the same manner, a satisfy condition (0/0) in the feeders actually indicates the presence of a carry when refering to true operands. Thus, the definition of a carry has been reversed and causes the result to be complemented "automatically" within the adder. With referrence to the true values of Xj and $\overline{\text{Xk}}$ (not the contents of the feeders), the formulas defining sums of 1 and 0 become:

1) $\overline{\text{EQUIVALENCE}} \cdot \text{CARRY IN} + \text{EQUIVALENCE} \cdot \overline{\text{CARRY IN}} = 1$

2) $\text{EQUIVALENCE} \cdot \text{CARRY IN} + \overline{\text{EQUIVALENCE}} \cdot \overline{\text{CARRY IN}} = 0$

The student is now referred to Figure 7.6, where the Xj - Xk adder is representatively shown. To the left are the QA modules which contain the feeder registers. Next are QB modules which check for stage equivalence and carries entering stages within a 3-bit group.

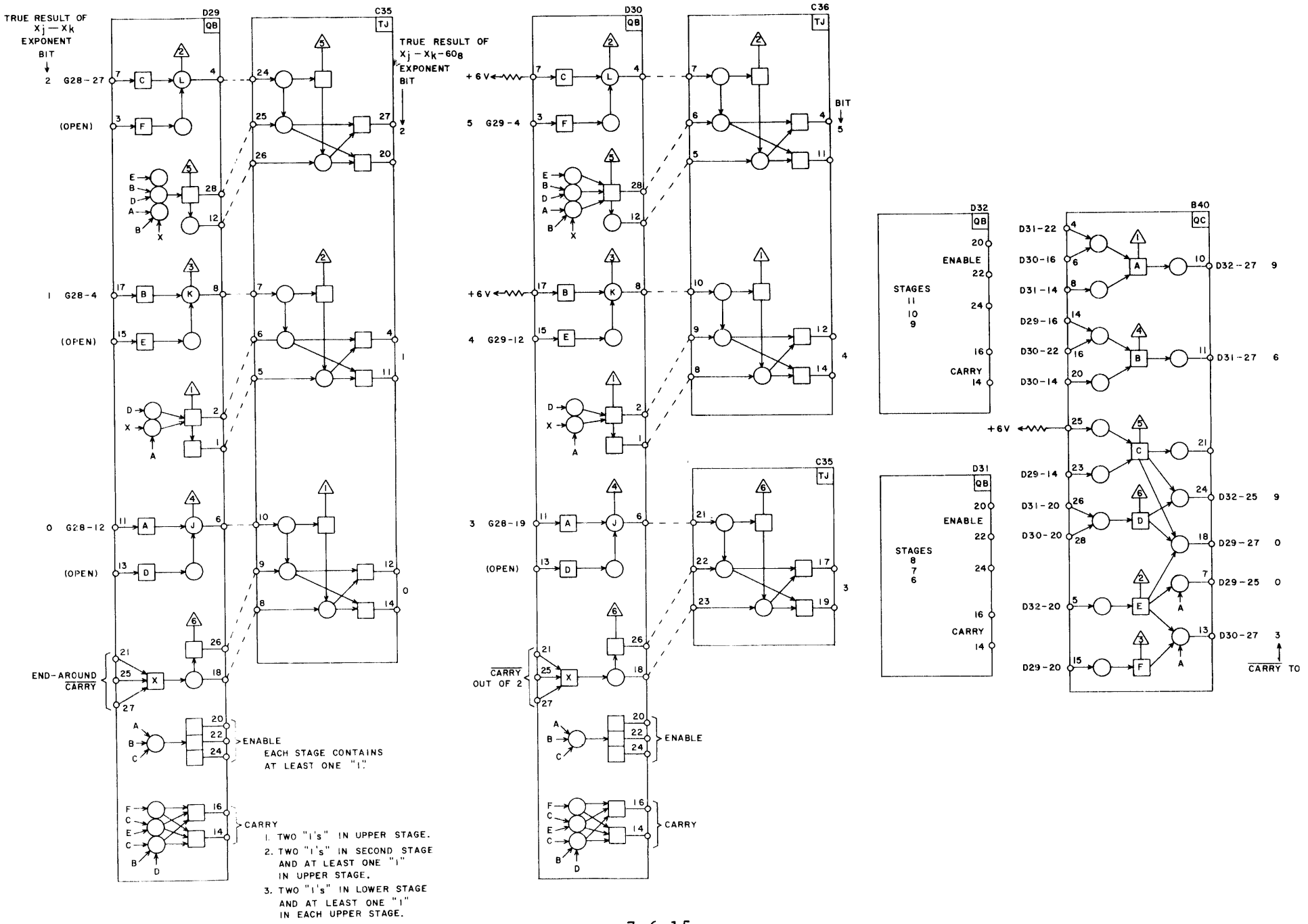
For example, test points 4, 3 and 2 indicate "equivalence" in stages 2^0, 2^1 and 2^2, respectively. Also, test points 6, 1 and 5 indicate the "carry in" condition to stages 0, 1 and 2 respectively. These conditions are combined on the TJ modules according to formulas #1 and #2 above. To the right are shown more QB modules and QCs which determine which groups have carries in. This determination is made by combining the carry out (generate) and enable conditions of individual stages. The procedure used for carry checks is similar to that used in other 6600 adders and is therefore not belabored. If detailed logic analysis is required, the Chassis #2 wiring tabs should be used.

The true value of the result is seen at the output pins of the TJ modules and is used as an input to the Xj - Xk - $60_{(8)}$ adder.

Xj - Xk - $60_{(8)}$ ADDER

Refer to Figure 7.6-15 during the following discussion. Since QB and TJ modules are used in this adder, as in the Xj - Xk adder, the concept of the two adders must be the same; in other words, the result is generated by adding complements. No feeder registers are required, since one input is the output of adder #1 and the other input is always $60_{(8)}$ or $000\ 000\ 110\ 000_{(2)}$. To understand the wiring of the inputs to this adder, the values that the QB modules "look for" should be kept in mind. For example, in the Xj - Xk adder the following translations for various QB terms are made (with relation to the feeders):

SUBTRACTS 60_8 BY ADDING COMPLEMENT



Term J ⇒ EQUIVALENCE (i.e. $Xj \cdot \overline{Xk} + \overline{Xj} \cdot Xk$)
Term A ⇒ $\overline{Xj} \cdot Xk$ (or, $Xj + Xk$)
Term D ⇒ $Xj \cdot Xk$

Since one of the inputs to the Xj - Xk - $60_{(8)}$ adder is $000\ 000\ 100\ 000_{(2)}$ ($60_{(8)}$) terms D, E and F on D29 (bits 2^0-2^2) and term D on D30 (bits 2^3-2^5) can be forced to a "0" (since two ones are never possible in the first four or upper six stages).

On the other hand, terms A, B, and C on D29 and term A on D30 should output a "1" when the corresponding bit of the result, Xj - Xk, is a "zero". This is justified, since if a feeder register were used, it would hold a "1" when the result was actually a "zerp" (since it would hold the complement). Because the second input is always $000\ 000\ 110\ 000_{(2)}$ the condition of two zeros in bits 0-3 and 6-11 occurs when ever the corresponding bit of the sum, Xj and Xk, is a one. Hence, the following translations are used:

D29, term A⇒$\overline{20}$
D 29,term B⇒$\overline{21}$
D29, term C⇒$\overline{22}$
etc.

Note that two "1"s is also the condition which generates a "carry" (see the inputs to TP1).

The bit 2^4 and 2^5 logic differs somewhat from the other bits since one input to these stages is always a "one" (i.e. $60_{(8)}$). In this case, two "zeros" in the feeders is never possible and terms B and C on D30 may be forced to ones. This is done by connecting the input pins to +1.2 volts. Terms E and F and now enabled and will output a 1 if 2^4 or 2^5 respectively is a "zero". Thus, the outputs of test points 4, 3 and 2 on the QB modules indicate the "Equivalence" condition and the outputs of test points 6, 1 and 5 the "carry in" condition. Carry

in checks and final equivalence/carry summation are identical to the Xj - Xk adder and are, therefore, not discussed further.

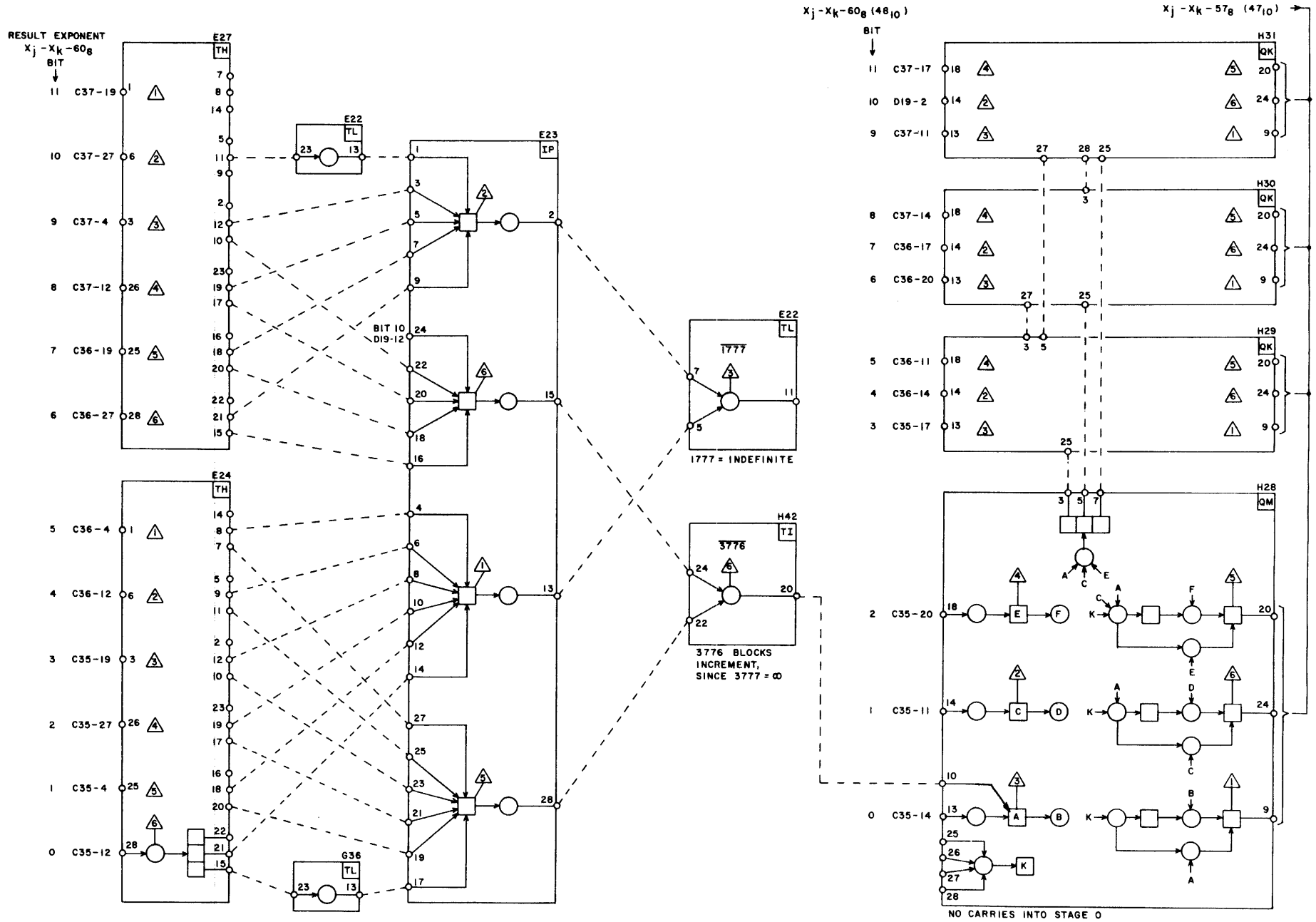
PLUS ONE NETWORK

This network, shown in Figure 7.6-16 adds one to the output of adder #2. This is accomplished on QM and QK modules. (Note: With the exception of the pin 10 input, the QM module is identical to the QK module).

The incrementer can be considered as being divided into four 3-bit groups, one group per module. In order for a group to be incremented, there must be a carry in from the preceding groups. Since end around carries do not exist, group zero is an exception. Because the junction of this circuit is to add 1 to the value Xj - Xk - $60_{(8)}$, group zero always has a carry-in (term K is always a "one" since the input pins, 25,26,27, & 28 are open). Thus, the incrementing of group zero is always enabled since term K = 1 is the requisite for incrementing a group.

Incrementing group 1 (H29) occurs if group 0 is all "ones" ($7_{(8)}$), incrementing group 2 (H30) occurs if both groups 0 and 1 are all "ones" ($77_{(8)}$), and incrementing group 3 (H31) occurs if groups 0, 1, and 2 are all "ones" ($777_{(8)}$).

Incrementing within a group occurs as follows. If term K is a "1", (i.e. there is a carry into the group) the least significant bit is toggled. The next significant bit is toggled if the least significant bit is a "one". The most significant bit is toggled if both bits of


RESULT EXPONENT
$X_j - X_k - 60_8$
BIT
E27
TH
E22
TL
E23
IP
BIT 10
D19-12
E24
TH
G36
TL
E22
TL
1777
1777 = INDEFINITE
H42
TI
3776
3776 BLOCKS
INCREMENT,
SINCE 3777 = ∞
$X_j - X_k - 60_8$ (48_{10})
BIT
H31
QK
H30
QK
H29
QK
H28
QM
$X_j - X_k - 57_8$ (47_{10})
NO CARRIES INTO STAGE 0

lesser significance are "ones". This can be proved by studying the logic shown in Figure 7/6-16. In this manner, the value $Xj\text{-}Xk\text{-}60_{(8)}$ is incremented by one to form $Xj\text{-}Xk\text{-}57_{(8)}$, which is used as the final exponent if normalization of the quotient coefficient is required.

Very little timing is required for generation of the exponents since most of the exponent logic is static. The exponents of Xj and Xk are gated to the exponent feeders at the same time the coefficients are sent to their feeders (See Section 7.6.4). From that point on, the exponent bits filter through the adders discussed above and through the test logic discussed in Section 7.6.9. When a "Transmit" is received from the scoreboard, the final exponent is gated on the data trunk.

7.6.9 POPULATION COUNT CONTROL

The population count logic of the Divide unit counts the number of "1" bits in operand register Xk and stores the count in operand register Xi. Since $60_{(10)} = 74_{(8)}$, only the lower six bits of the chassis 2 output network are required to transmit the result to Xi. Since the time required for generation of the 6-bit population count (800 nanoseconds) is considerably less than the 2.9 microseconds needed for a floating point divide, and the process itself is different, separate control logic is utilized.

Figure 7.6-20 shows the population count control logic and should be used in conjunction with the timing chart (Figure 7.6-21) during the following discussion.

When the population count instruction (fm=47) is issued to the scoreboard, the "Pop. Count" mode bit is sent to chassis 2 and received by Test Point 3 on J02. After resolving any second order conflicts which may exist, the Scoreboard sends the "Go Divide" signal which initiates the Divide timing chain. (See Section 7.6.4), and sets the "Go Pop. Count" flip-flop (D27,TPZ). This enables setting D27, TP5 (t225) and clearing the Counter network feeder registers (t240). D27, TP5 enables the "Gate In" signal which enters operand Xk from the chassis 2 Input register to the counter network feeder register.

The first step in generating the Population Count is to generate "4-bit sums". (See Section 7.6-13 for logic analysis of the Population Count Network). This logic is static (i.e. no gate is required), so term #6 on the timing chart indicates the time allowed for generation of the 4-bit sums (The time lapse from entering the feeders until the

first sum is generated).

The fifteen 4-bit sums are added together using four iterations each of which is 100 nanoseconds in duration. The iterations are shown with terms 7, 8, 9, and 10. The solid pulses illustrate gating into the Adder feeder registers on t50, while the slashed pulses show the stages of the divide timing chain that enable each of the four iterations.

The "Request Release" is gated from chassis 2 at approximately t540 if the Divide unit is in the Pop. Count mode (term #11). The divide timing chain is disabled on the t25 after setting the t650-750 stage in the chain (term 12). This is accomplished by disabling the setting of the t725-825 stage of the chain, thereby preventing further propagation to the remaining stages.

The result is gated to the result register upon receipt of the "Transmit" signal from the scoreboard. The earliest possible time (no third order conflicts) is t740 as illustrated by term #13. Only transmitters fro bits 2^0-2^5 are used during the Pop. Count mode, since the count cannot exceed $74_{(8)}$.

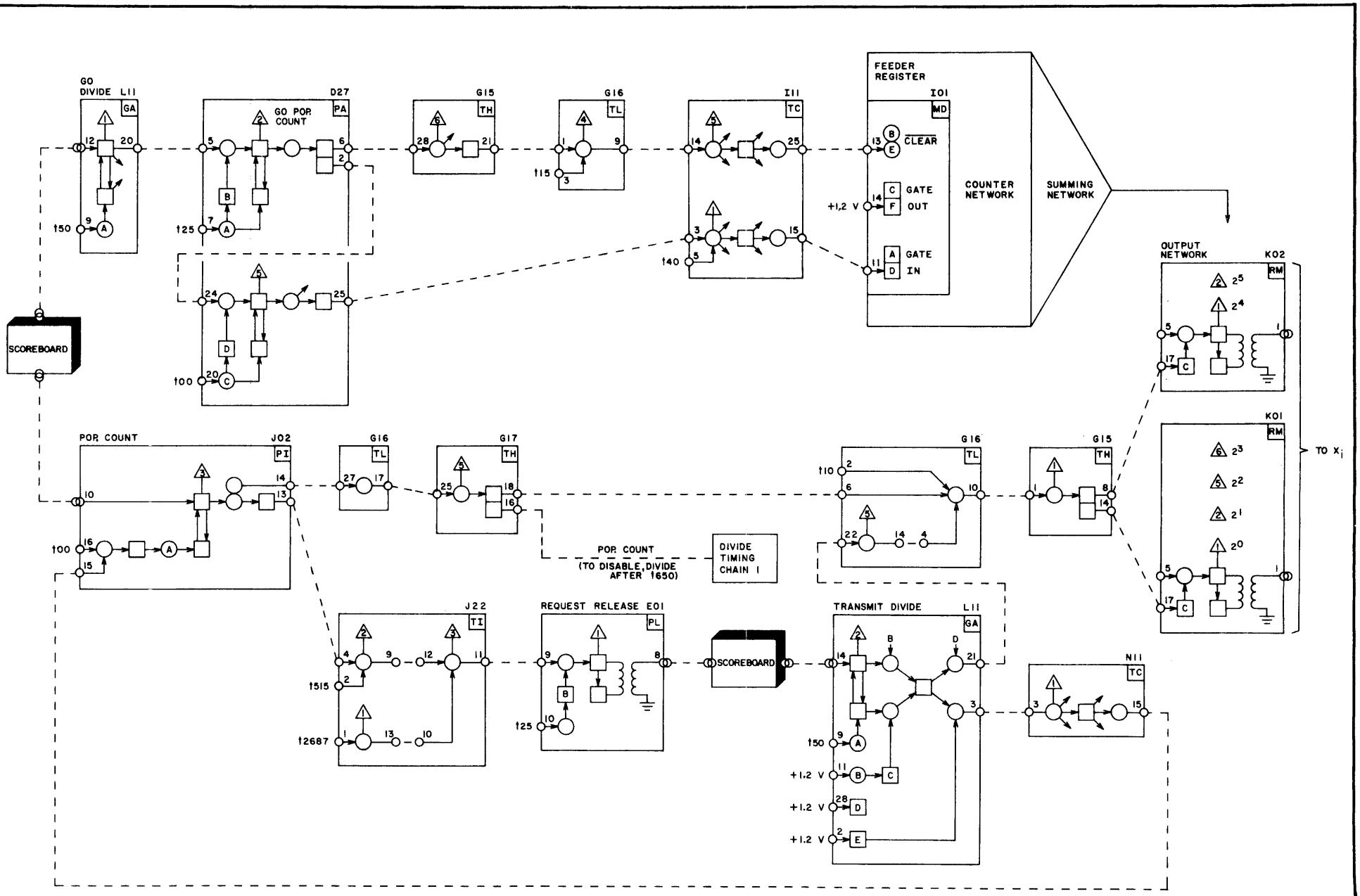
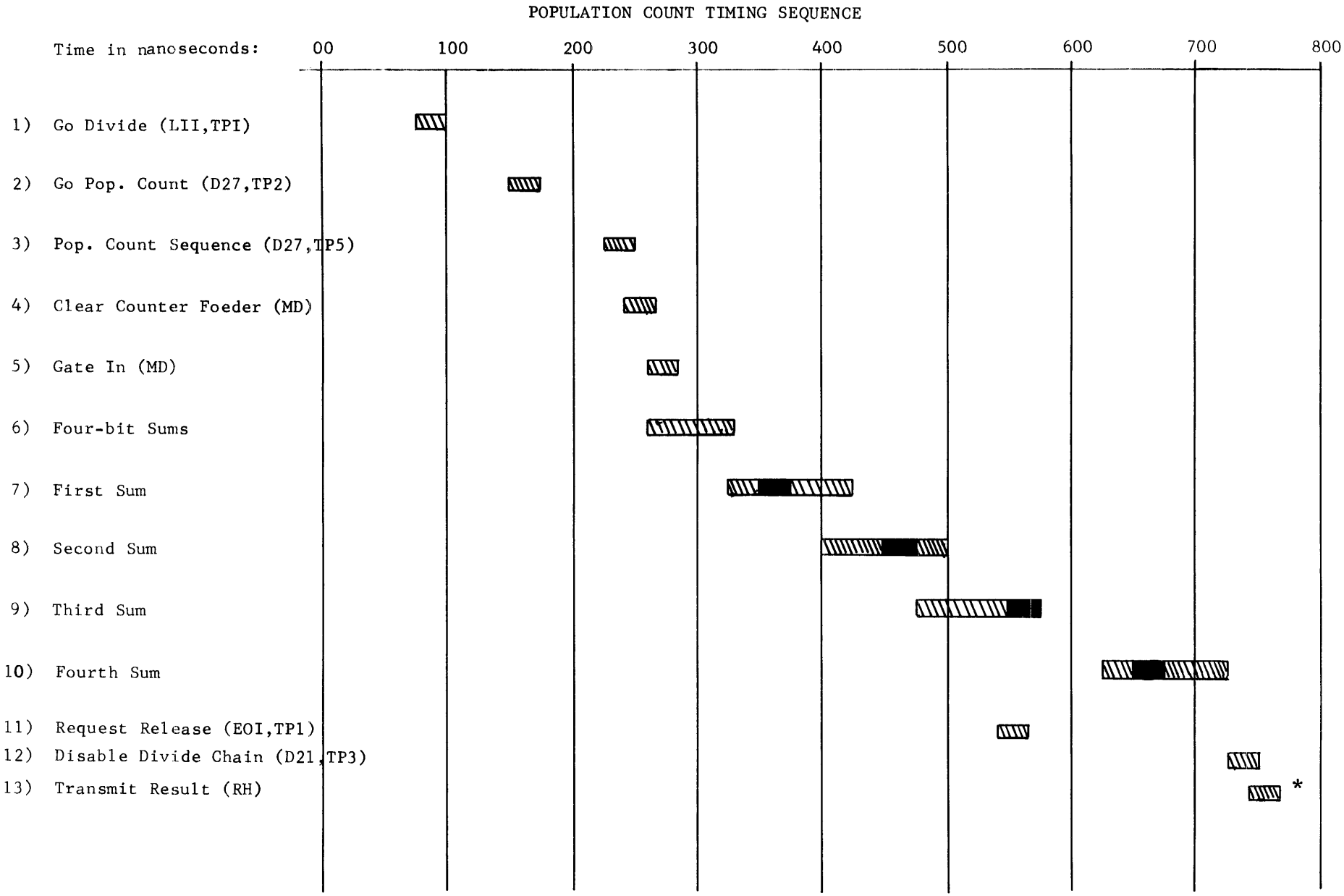

GO DIVIDE L11
GA
GO POR COUNT
D27
PA
G15
TH
G16
TL
I11
TC
FEEDER REGISTER
I01
MD
CLEAR
GATE OUT
GATE IN
+1,2 V
COUNTER NETWORK
SUMMING NETWORK
OUTPUT NETWORK
K02
RM
K01
TO x_j
SCOREBOARD
POR COUNT
J02
PI
G17
POR COUNT (TO DISABLE DIVIDE AFTER 1650)
DIVIDE TIMING CHAIN 1
J22
TI
REQUEST RELEASE E01
PL
TRANSMIT DIVIDE
N11

Figure 7.6-20



* Earliest possible time - no third order conflicts

Figure 7.6-21

7.6.10 POPULATION COUNT NETWORK

Generation of the population count is accomplished in five steps, each of which requires 100 nanoseconds. This is illustrated in block diagram form in Figure 7.6-22.

1) Divide the 60-bit operand Xk into fifteen 4-bit groups and determine the number of "1's" in each group. The totals ("4-bit sums") for each group will be 0,1,2,3 or 4, and will therefore, be 3-bits long.
2) Combine the outputs of the 4-bit sum generators two at a time (in parallel) to generate the sum of "ones" in 8-bit groups. This is done by using eight add networks. Seven of the networks generate sums of eight bits and the eighth, the sum of the upper 4-bits (which is already available from the upper 4-bit sum generator, but is filtered through the add network for timing purposes.) This iteration is referred to as the "First Sum".
3) Combine the outputs of the first sum adders two at a time (in parallel) by using four add networks. Three of the networks generate the sum of "ones" in 16-bit groups and the fourth, the sum of 12 bits. This is referred to as the "Second Sum."
4) Combine the outputs from the second sum adders two at a time (in parallel) by the use of two add networks. One network generates the sum of "ones" in 32-bits and the other, the sum of 28-bits. This is the "Third Sum".
5) The "Fourth Sum" adds the outputs of the third sum adders to form the final count of "ones" in the 60-bits of Xk. Since the count cannot exceed $74_{(8)}$, only the lower six transmitters of

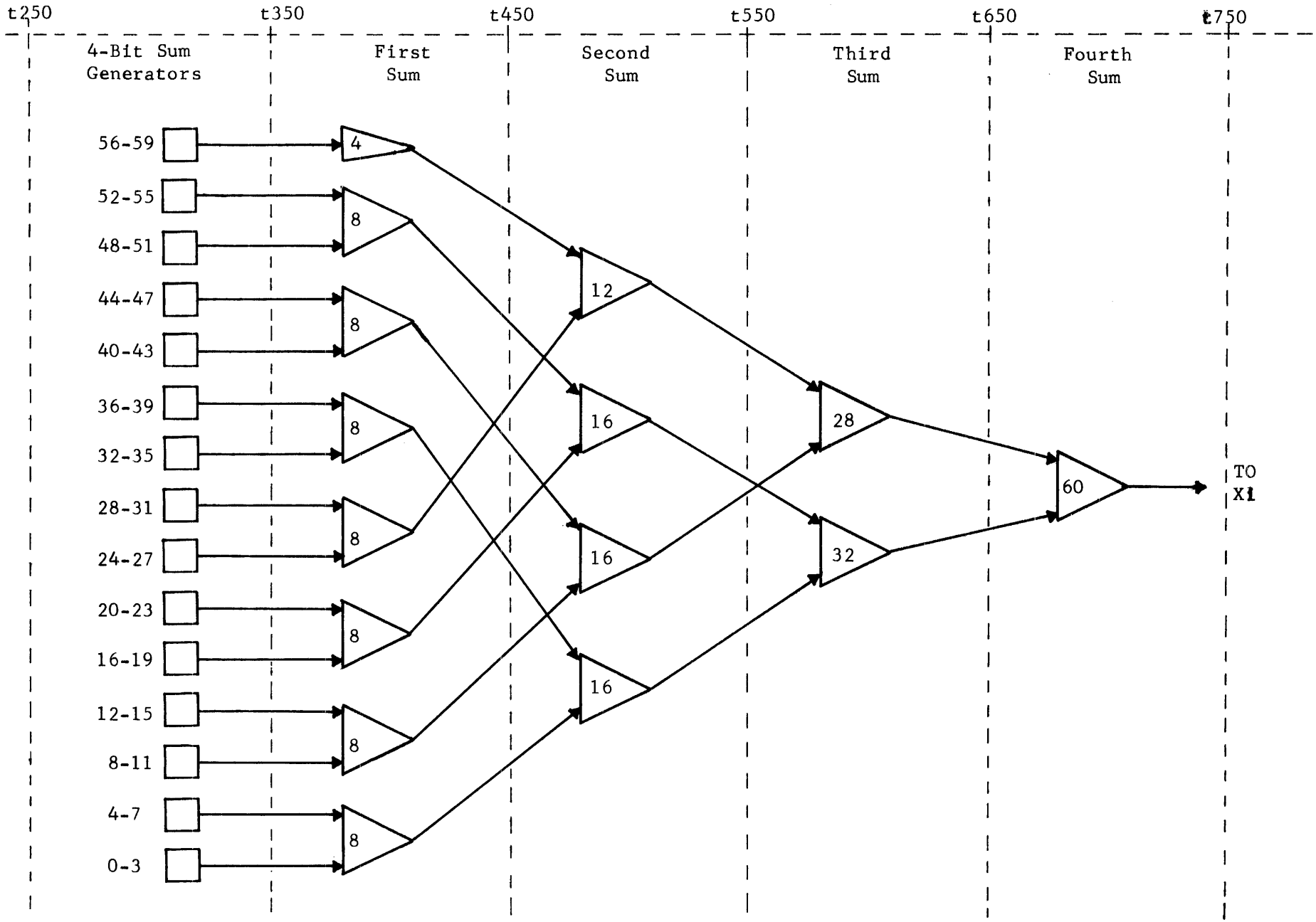

t250
t350
t450
t550
t650
t750
4-Bit Sum Generators
First Sum
Second Sum
Third Sum
Fourth Sum
56-59
52-55
48-51
44-47
40-43
36-39
32-35
28-31
24-27
20-23
16-19
12-15
8-11
4-7
0-3
4
8
8
8
8
8
8
8
12
16
16
16
28
32
60
TO X1

Figure 7.6-22

the data trunk are used in gating to Xi.

Figure 7.6-23 shows the logic required for one 4-bit sum generator. To the left is the feeder register which is held on MD modules. The feeder contents are sent to KD modules, each of which generates a 3-bit long count of ones for a 4-bit group. The first check made on the KD's is for equivalence or $\overline{\text{equivalence}}$ between two adjacent bit positions. The existance of equivalence implies that the two bits position contain 0 or 2 "ones". $\overline{\text{Equivalence}}$ indicates that only one "one" exists in the two bits. H01, TP4 translates as:

$$\overline{2^0}\cdot 2^1 + 2^0\cdot\overline{2^1} \quad \text{1 "one"}$$

and term F as:

$$2^0\cdot 2^1 + \overline{2^0}\cdot\overline{2^1} \quad \text{0 or 2 "ones"}$$

Test point 1 makes the same translation for bits 2^2 and 2^3. These outputs are combined to determine the number of "ones" in 4-bit positions. The following translations are made for test points 6, 5 and 2, respectively, of the sum of ones: The numbers in the formulas indicate bit positions. The following translations are made for test points 6,5 and 2, which are bits o, 1, and 2, respectively, of the sum of ones: The numbers in the formulas indicate bit positions.

$$\begin{aligned}\text{TP6}\quad & 0{:}\overline{1}\cdot\underline{2}\cdot 3 + 0\cdot\overline{1}\cdot\overline{2}\cdot\underline{3} + \underline{0}\cdot 1\cdot\underline{2}\cdot 3 + \underline{0}\cdot 1\cdot 2\cdot\underline{3}+\\ & 0\cdot 1\cdot\overline{2}\cdot 3 + 0\cdot 1\cdot 2\cdot\overline{3} + \underline{0}\cdot\overline{1}\cdot\overline{2}\cdot 3 + \underline{0}\cdot\overline{1}\cdot 2\cdot\overline{3}\end{aligned}$$

The formula covers all possible one/zero configurations which give an "odd" sum of ones (1 or 3), since only then should 2^0 be set.

$$\begin{aligned}\text{TP5}\quad & (\overline{0\cdot 1})(\underline{2\cdot 3}) + (\overline{0\cdot 2})(\underline{1\cdot 3}) + (\overline{0\cdot 3})(\underline{1\cdot 2})+\\ & (0\cdot 3)(\overline{1\cdot 2}) + (0\cdot 2)(\overline{1\cdot 3}) + (0\cdot 1)(\overline{2\cdot 3})\end{aligned}$$

This formula includes all possible bit combinations that yield a sum of 2 or 3 "ones", in which case bit 2' should be set

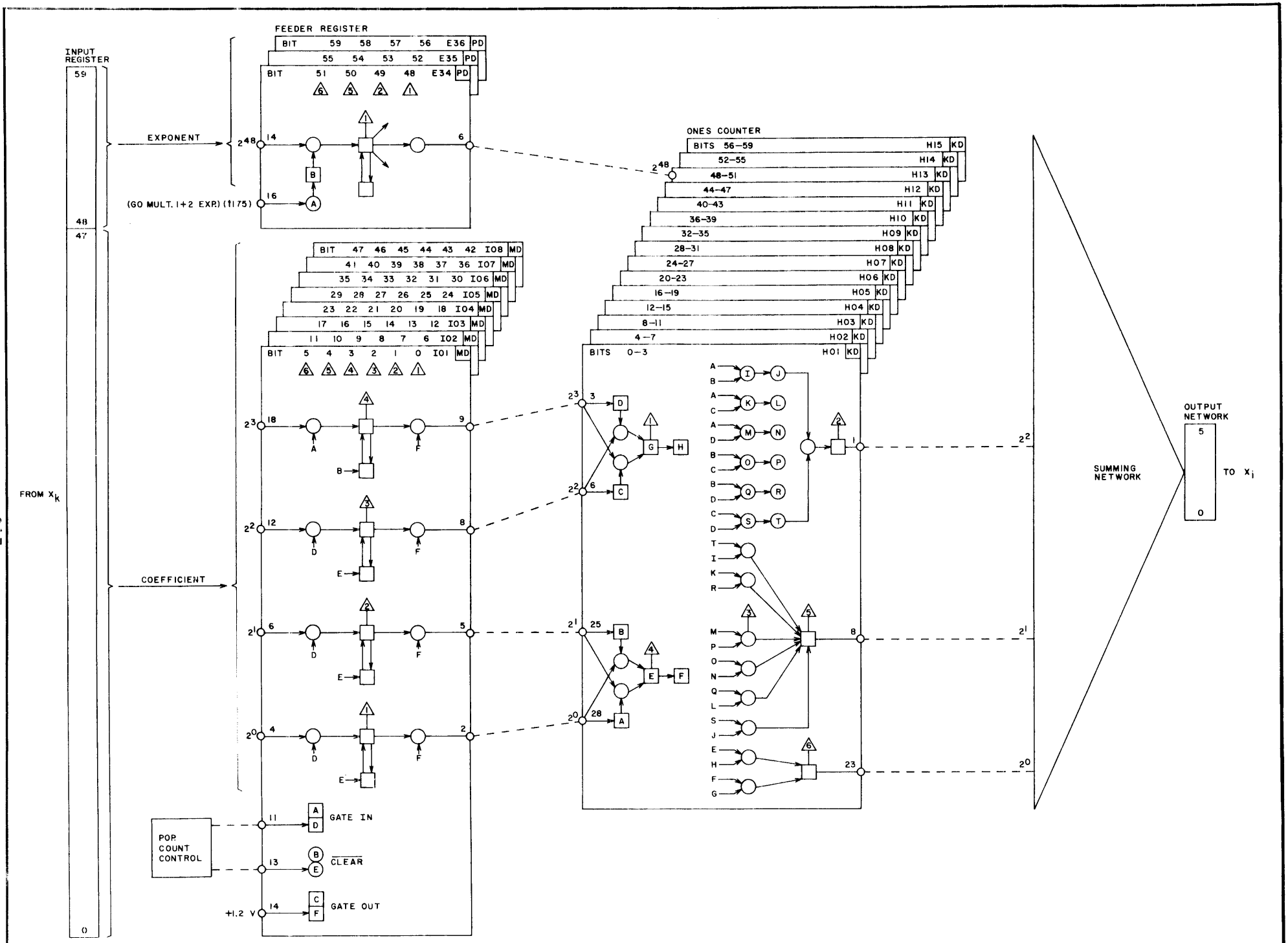

INPUT REGISTER
59
48
47
0
FROM X_k
EXPONENT
COEFFICIENT
FEEDER REGISTER
BIT 59 58 57 56 E36 PD
55 54 53 52 E35 PD
BIT 51 50 49 48 E34 PD
(GO MULT. 1+2 EXP) (1175)
BIT 47 46 45 44 43 42 I08 MD
41 40 39 38 37 36 I07 MD
35 34 33 32 31 30 I06 MD
29 28 27 26 25 24 I05 MD
23 22 21 20 19 18 I04 MD
17 16 15 14 13 12 I03 MD
11 10 9 8 7 6 I02 MD
BIT 5 4 3 2 1 0 I01 MD
ONES COUNTER
BITS 56–59 H15 KD
52–55 H14 KD
48–51 H13 KD
44–47 H12 KD
40–43 H11 KD
36–39 H10 KD
32–35 H09 KD
28–31 H08 KD
24–27 H07 KD
20–23 H06 KD
16–19 H05 KD
12–15 H04 KD
8–11 H03 KD
4–7 H02 KD
BITS 0–3 H01 KD
POP COUNT CONTROL
GATE IN
CLEAR
GATE OUT
+1.2 V
SUMMING NETWORK
OUTPUT NETWORK
TO X_i

Figure 7.6-23

TP2 0·1·2·3

This is the only possibility that yields a sum of 4 and it will therefore set bit 2^2 of the sum.

These translations are made for each 4-bit group and are subsequently added together in the summing network.

The summing network is a standard full adder which performs all four additions required to form the pop. count. This is accomplished by using four iterations. (See Section 7.5-12 for the Pop. count timing sequence). Since each iteration requires fewer inputs than the previous, each will use less stages of the adder. (Refer to the C. E. Deagrams, sheet 210). The first sum adder networks have 3-bit inputs from the 4-bit sum generators. During the second iteration, 4-bit inputs are used since the value of the just sum may be as high as $8_{(8)}$, of $10_{(8)}$. For the third and fourth iterations 5-bit inputs are required since the second and third sums may be as high as $20_{(8)}$ and $40_{(8)}$, respectively.

The same add network is used for each of the four iterations of the population count, although all of the inputs are not used for the second, third and fourth iterations since the number of inputs become successively smaller for each iteration.

Figure 7.6-24 shows the logic for three bits of the summing network. The TE modules are used to fan-in and select the inputs during each iteration. On the module shown, only enables for the first three sums are shown since the add network fed by the F09 QA module (feeders) is used only during the first three iterations. F11 (sheet 210) is the

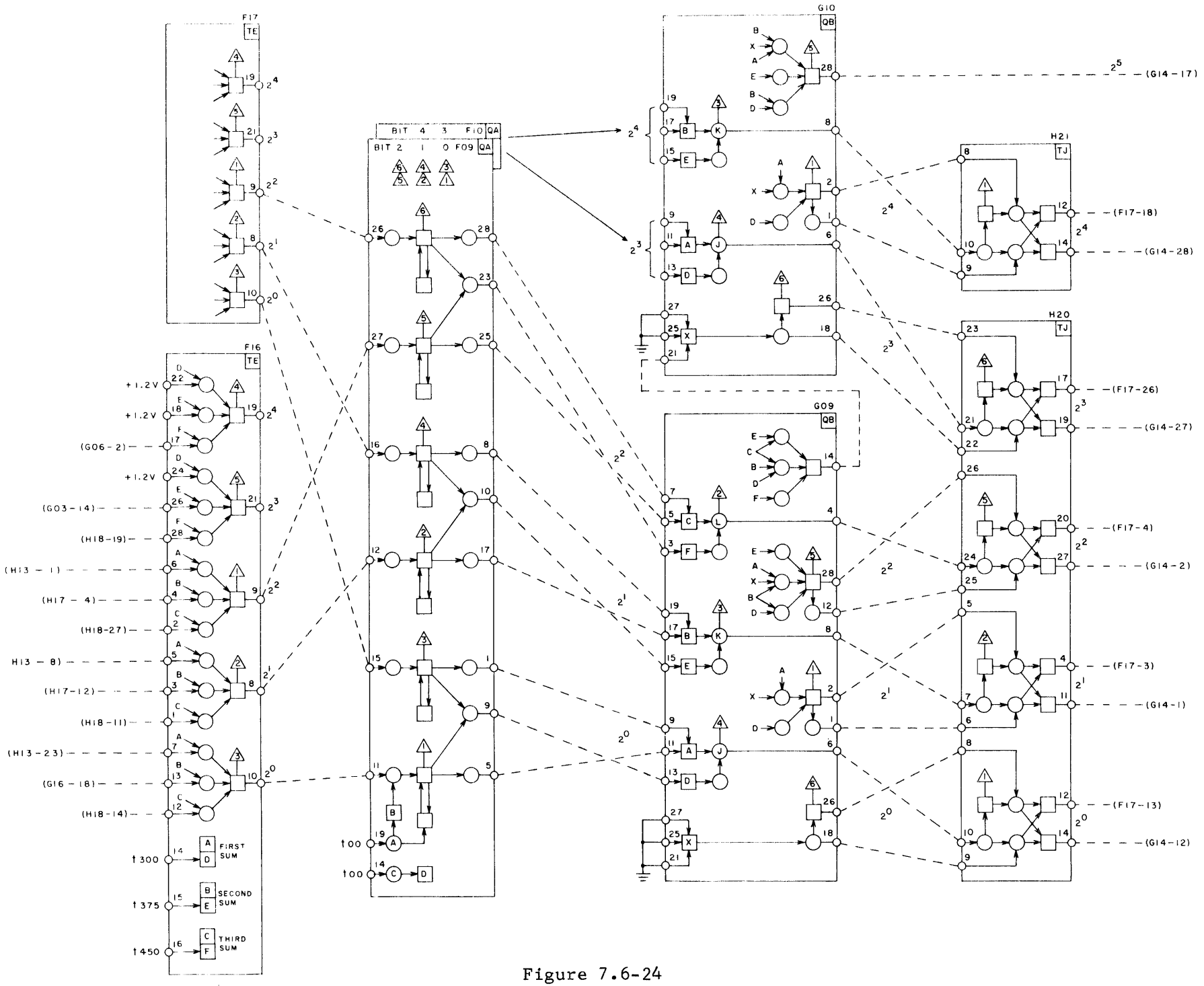

F17
TE
F16
TE
BIT 4 3 F10 QA
BIT 2 1 0 F09 QA
G10
QB
G09
QB
H21
TJ
H20
TJ
+1.2V
+1.2V
(G06-2)
+1.2V
(G03-14)
(H18-19)
(H13-1)
(H17-4)
(H18-27)
H13-8)
(H17-12)
(H18-11)
(H13-23)
(G16-18)
(H18-14)
t300
t375
t450
t00
t00
FIRST SUM
SECOND SUM
THIRD SUM
(G14-17)
(F17-18)
(G14-28)
(F17-26)
(G14-27)
(F17-4)
(G14-2)
(F17-3)
(G14-1)
(F17-13)
(G14-12)

Figure 7.6-24

only QA module used during all four iterations. Therefore, the TE module feeding F11 will have four enables rather than three.

Each of the add networks operates essentially the same, using QA, QB, and Tj modules, so only one will be discussed in detail. The QA module feeders hold the true values of the inputs. The outputs of the feeders feed QB modules which perform two main functions.

1) Check for stage equivalence between the two source counts.
2) Check for and propagate carries from stage to stage to determine which stages have carries in.

The equivalence check outputs are on G09, test points 4, 3, and 2 for stages 0, 1 and 2 respectively. These translate as "Equivalence" and are sent to the TJ modules which perform the final carry/equivalence summation. Carries are propagated to the TJ modules, stages 1,2, and 3, from pins 2, 28 and 44 of G09 (stage 0 will never have a carry in). The TJ modules contain an equivalence network for each stage which checks for Equivalence and Carry In OR $\overline{\text{Equivalence}}$ and $\overline{\text{Carry In}}$ to yield a final sum of "1". The conditions Equivalence and Carry In OR $\overline{\text{Equivalence}}$ and Carry In result in a sum of "0" for that stage.

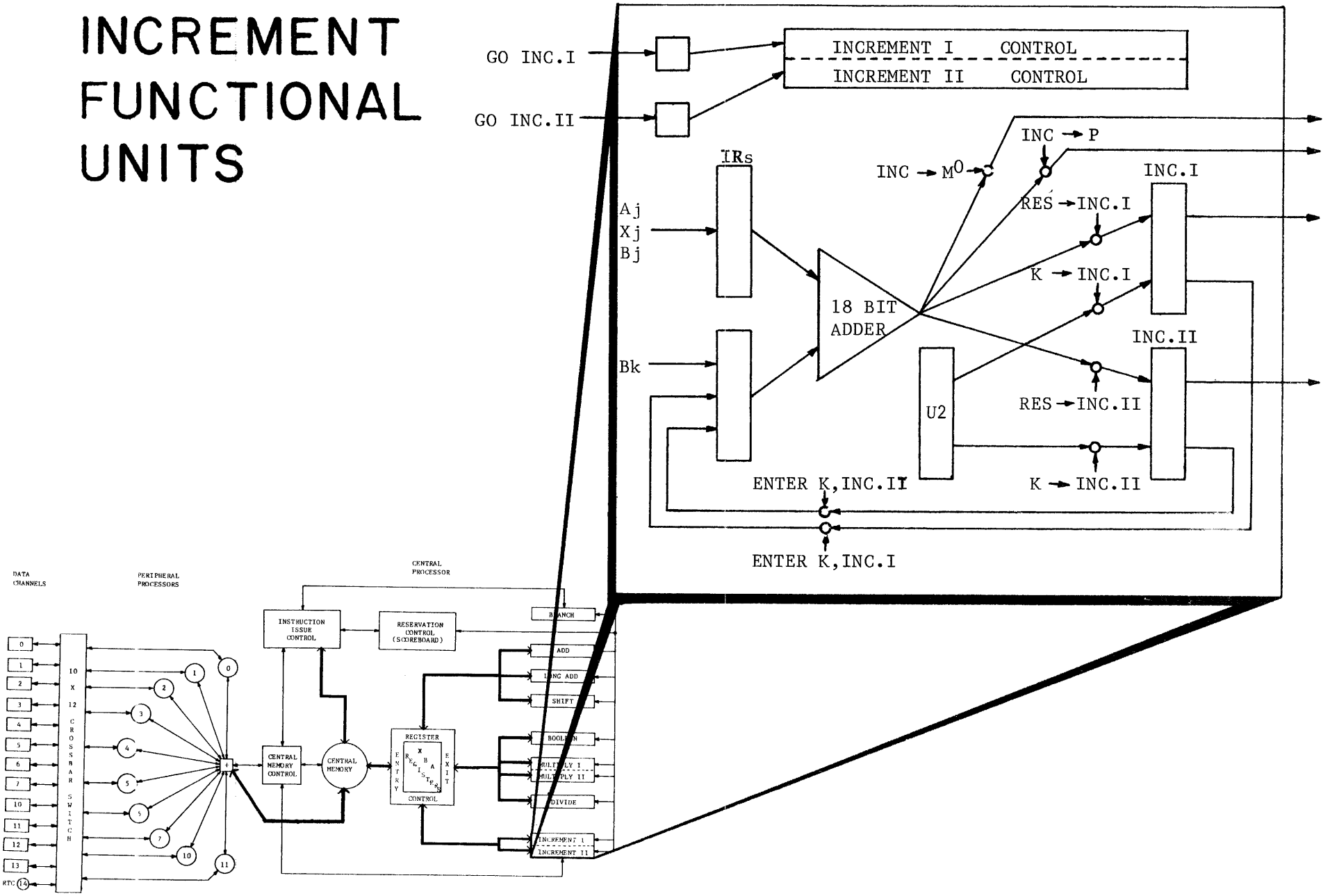
Detailed analysis of the gating on the TE modules is left to the student. The gating can be followed quite easily by using sheet 210 of the CE. Diagrams as a guide when tracing through the Chassis 2 wire tabs.

SECTION 7.7

INCREMENT

FUNCTIONAL UNITS

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 to accomplish 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, Bi + 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 - Bi = Bj

05 - Bi ≠ Bj

06 - Bi ≥ Bj

07 - Bi < Bj

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).

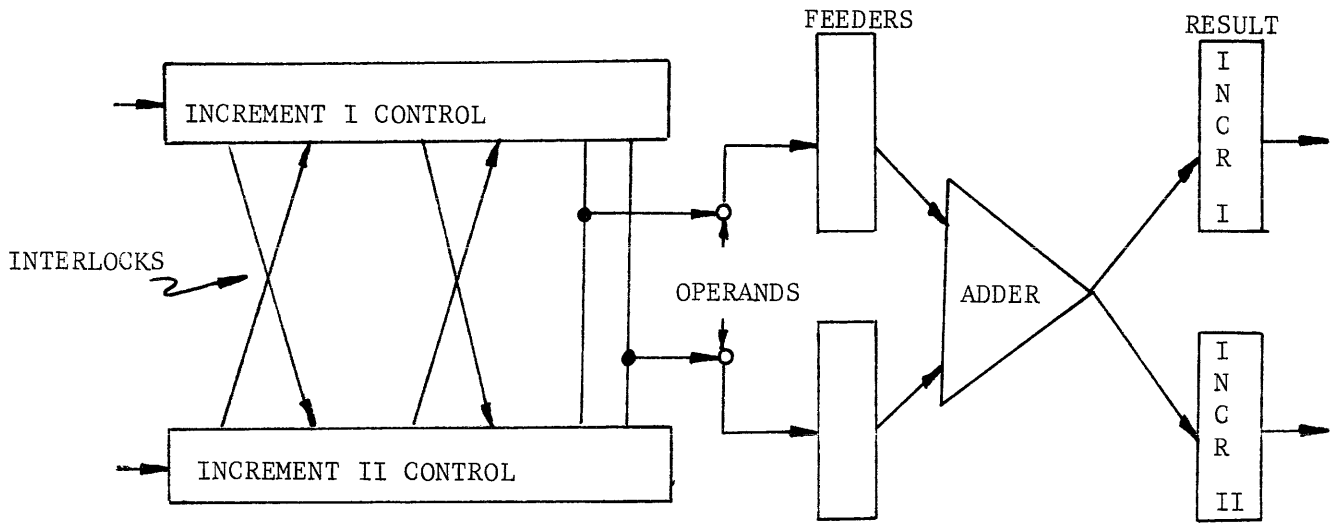


Figure 7.7-1

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 Ai

b) 60 - 67 - Result register is Bi

c) 70 - 77 - Result register is Xi

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 Aj and K to Ai	(30 bits)
51	SUM of Bj and K to Ai	(30 bits)
52	SUM of Xj and K to Ai	(30 bits)
53	SUM of Xj and Bk to Ai	(15 bits)
54	SUM of Aj and Bk to Ai	(15 bits)
55	DIFFERENCE of Aj and Bk to Ai	(15 bits)
56	SUM of Bj and Bk to Ai	(15 bits)
57	DIFFERENCE of Bj and Bk and Ai	(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, 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, Ai and Xi, by the 5 X 1 - 5 X 5 opcodes.

If i = 6 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 Ai, and Xi 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 fm) are sent from register Exist control to the Input registers (feeders) of the 18 bit adder. If fm = 50, 51, 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 (fm $\neq$ 50, 51, or 52) Bk is used

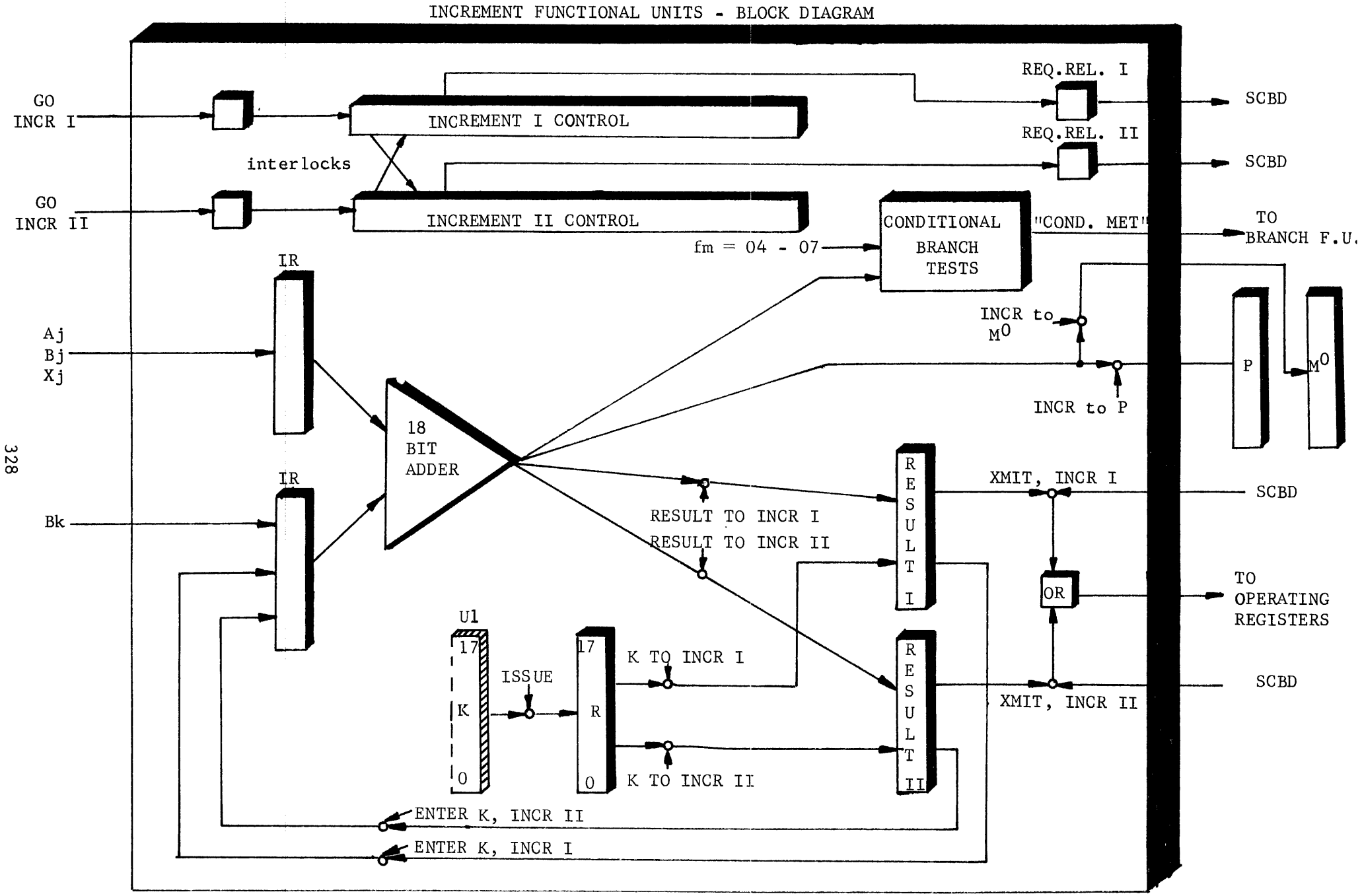

INCREMENT FUNCTIONAL UNITS - BLOCK DIAGRAM
GO INCR I
GO INCR II
interlocks
INCREMENT I CONTROL
INCREMENT II CONTROL
REQ.REL. I
REQ.REL. II
SCBD
SCBD
CONDITIONAL BRANCH TESTS
fm = 04 - 07
"COND. MET"
TO BRANCH F.U.
IR
Aj
Bj
Xj
IR
Bk
18 BIT ADDER
INCR to M0
INCR to P
P
M0
RESULT TO INCR I
RESULT TO INCR II
RESULT I
XMIT, INCR I
SCBD
OR
TO OPERATING REGISTERS
U1
17
K
0
ISSUE
17
R
0
K TO INCR I
K TO INCR II
RESULT II
XMIT, INCR II
SCBD
ENTER K, INCR II
ENTER K, INCR I

Figure 7.7-2

as the second operand. If fm = 55 or 57, Bk 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 X6 or X7 is stored in Central Memory.

6X Instructions

60	SUM of Aj and K to Bi	(30 bits)
61	SUM of Bj and K to Bi	(30 bits)
62	SUM of Xj and K to Bi	(30 bits)
63	SUM of Xj and Bk to Bi	(15 bits)
64	SUM of Aj and Bk to Bi	(15 bits)
65	DIFFERENCE of Aj and Bk to Bi	(15 bits)
66	SUM of Bj and Bk to Bi	(15 bits)
67	DIFFERENCE of Bj and Bk to Bi	(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 Bi = B0.

Data Flow (Refer to Figure 7.7-2)

The operand from Bj* and 18-bit constant K are placed in the feeder registers. Bj 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 0X instructions, U_i^1 and j are sent to U^2j and k respectively. Bj is thus designated by Bi of the original opcode and Bk by Bj.

04-07 Branch Instructions

04	GO TO K if Bi = Bj	(30 bits)
05	GO TO K if Bi $\neq$ Bj	(30 bits)
06	GO TO K if Bi $\geq$ Bj	(30 bits)
07	GO TO K if Bi $<$ Bj	(30 bits)

These instruction test an 18-bit word in register Bi against an 18-bit word in register Bj (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 Bj* and Bk* are sent to the Increment Unit feeder registers. For the conditional tests of equality (04 and 05) a bit by bit equivalence check is mode 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 (Bi - Bj) is examined.

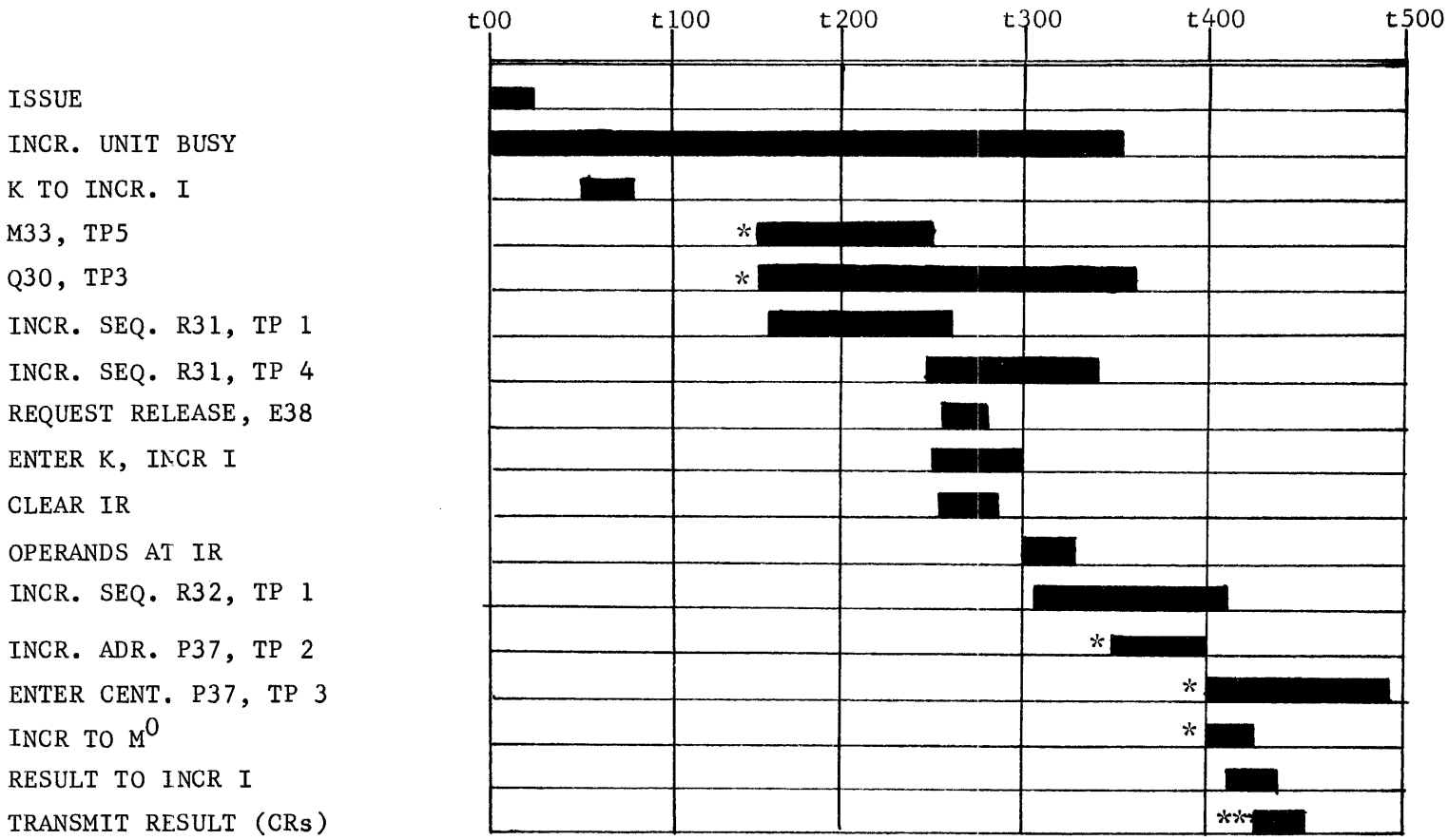
If it is zero, Bi $\geq$ Bj; if a one, Bi $<$ Bj. In all four cases, the result of the tests are combined with the opcode translation to generate the "Condition Met" signal (the absense of "Condition Met" implies "Condition Not Met") which is sent to the Branch functional unit to enable the branch sequence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

*Recall that on OX instructions, U^1i and j are sent to U^2j and k respectively. Bj is thus designated by Bi of the original opcode and Bk by Bj.

7.7.3 TIMING SEQUENCE

Figure 7.7-3 is a timing chart showing the sequence during increment operations. Terms are also included for operand read and write functions so, that orientation with respect to Stunt Box timing is possible. The page following the timing chart explains each term listed. It is assumed that Increment Unit I was selected.

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 Incremement Reuslt 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 adder 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 M^0
ENTER CENTRAL P37, TP3:	Set due to setting Incr. Adr. Flip/flop. Request priority to enter address into M^1 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 + Bi	M^0 and P registers
04 - 07	Condition Met or Not Met	Branch Unit
5X, i = 1-7	(A+B+X)j plus (Bk + K)	M^0 and Ai Registers
5X, i = 0	(A+B+X)j plus (Bk + K)	Ai register
6X	(A+B+X)j plus (Bk + K)	Bi register
7X	(A+B+X)j plus (Bk + K)	Xi 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.

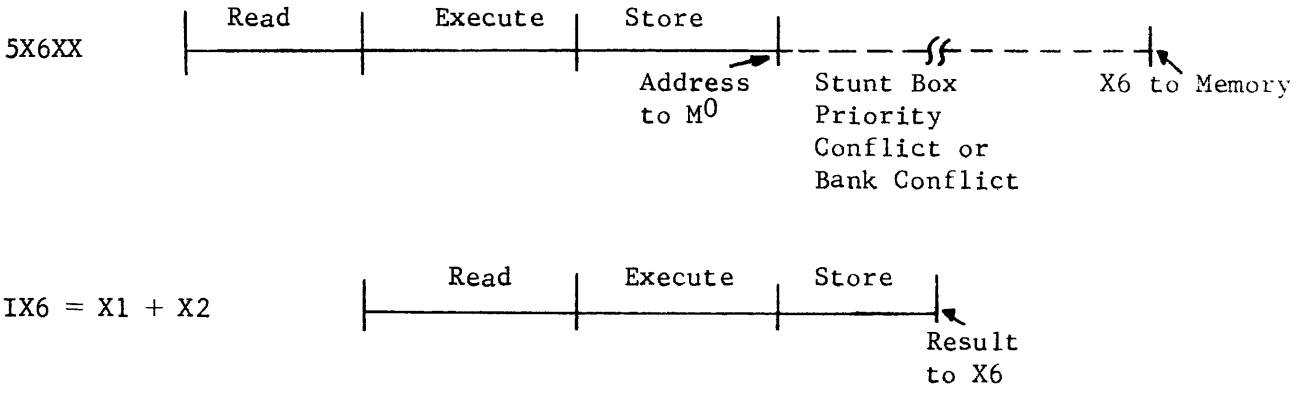
Increment First Order Conflicts (Special Cases): (5X)(i = 1-5)

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 Ai and Xi reservations will be checked (term T), and the scoreboard issue occurs only if both reservations are cleared.

The Ai and Xi reservations are cleared by separate gates, Ai is cleared upon "Release" of the Increment unit (Request Release and All Clear). Xi 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 cause second order conflicts 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)

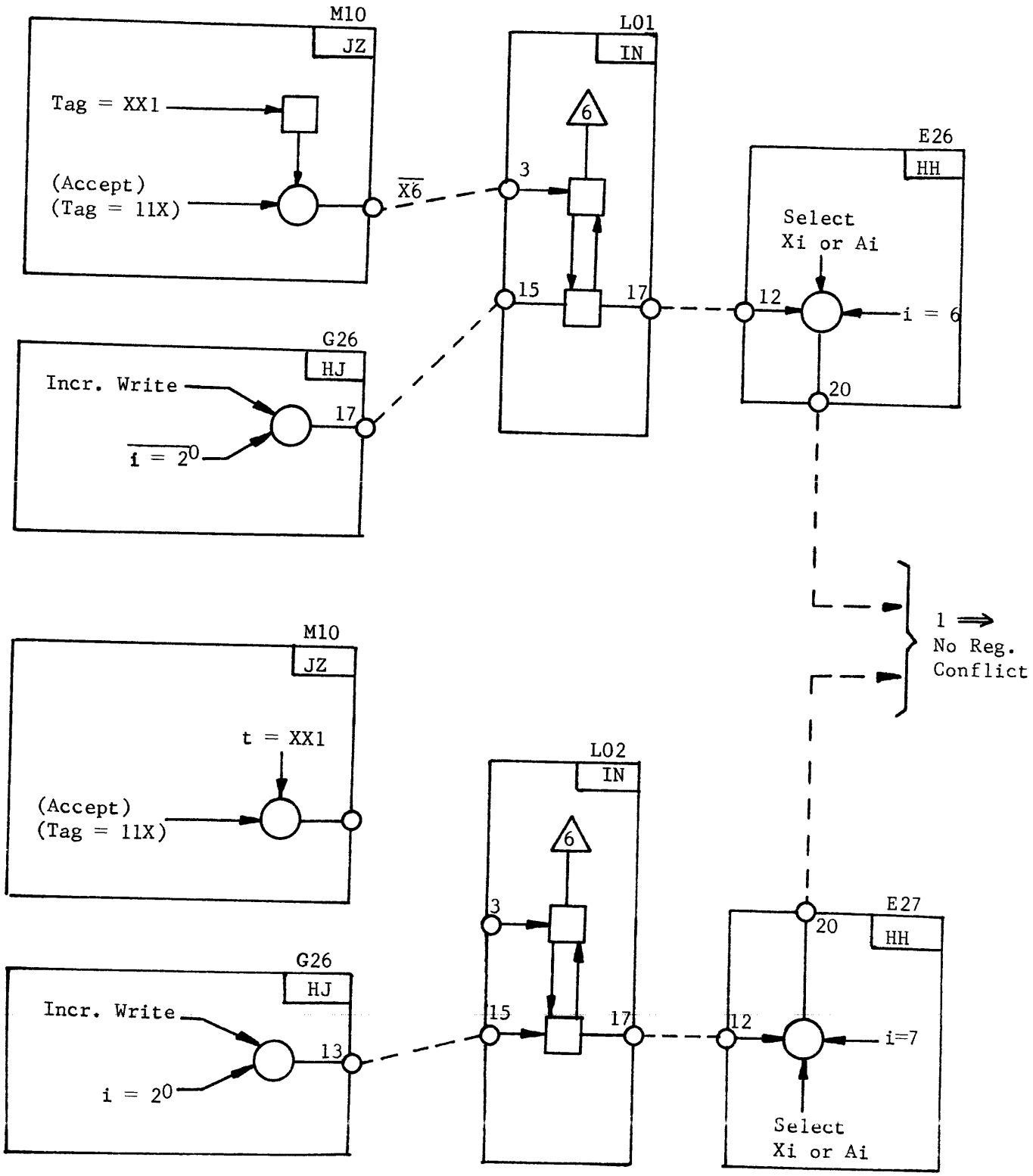


Figure 7.7-5

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 possiblility. Study the following example.

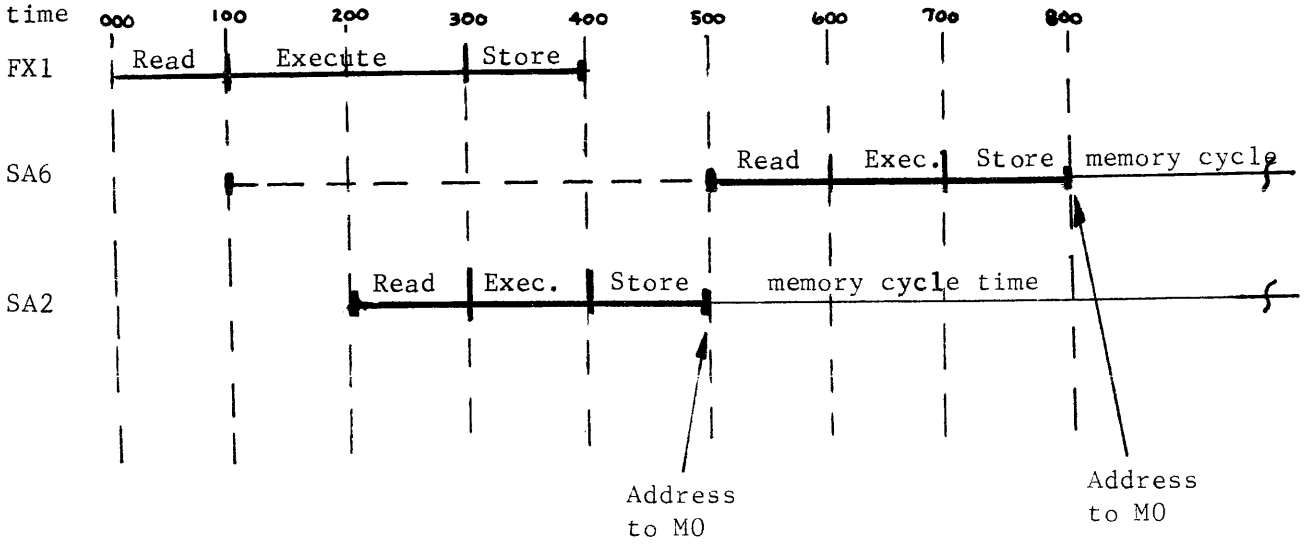
FX1 = X2 + X3 (30123)

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.

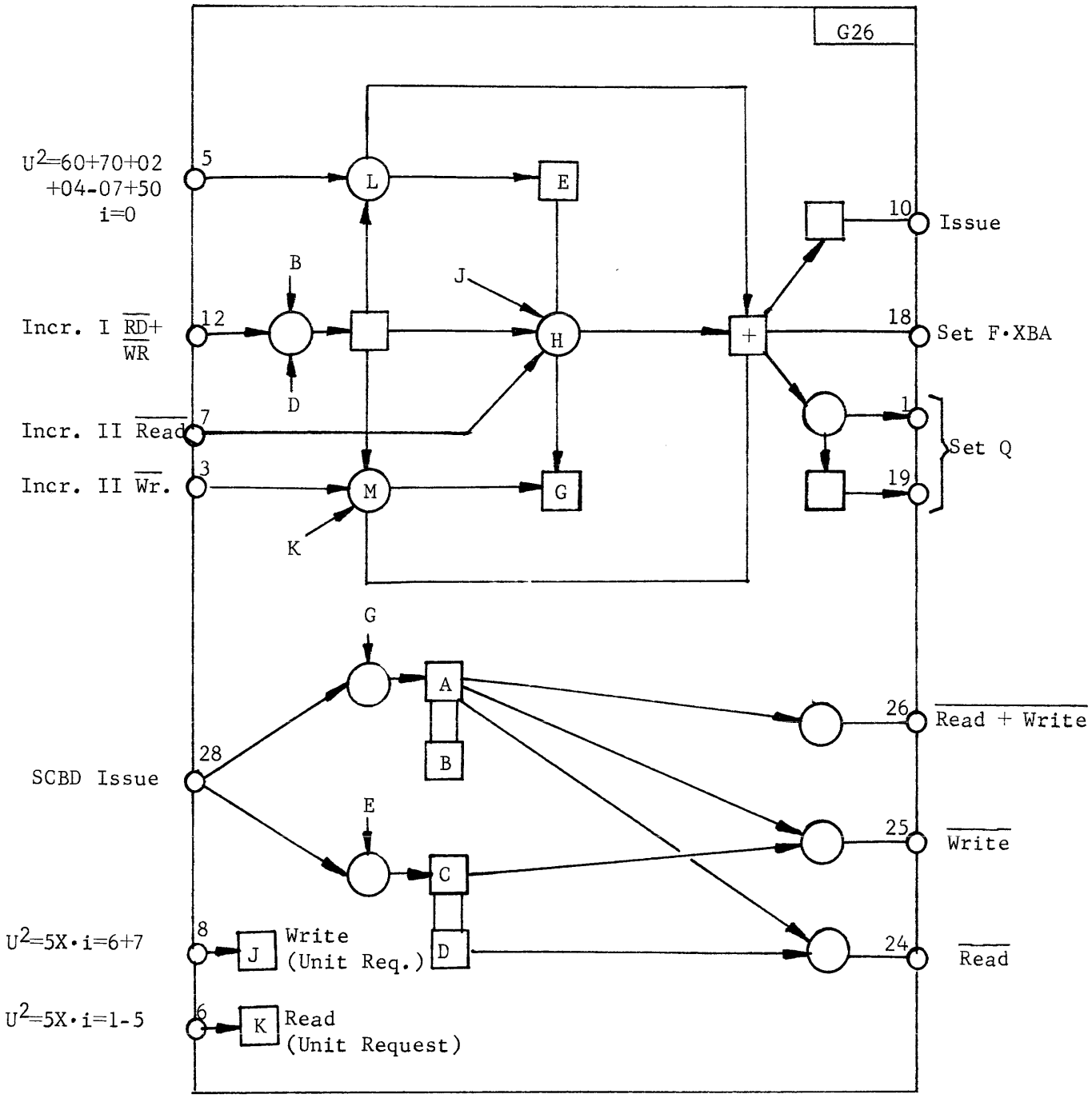
The Unit Request flip/flops (set on the U2 issue) have inputs on pins 6, 8, and 5. Term J indicates a Write unit request and term K a Read unit request. In order to generate the issue for a memory read request (term K) the AND gate, term M, must have all ones in. The translation for a zero out of M is: (Memory Read Request)(Incr. I Not Busy)(Incr. II $\overline{\text{Write}}$). 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

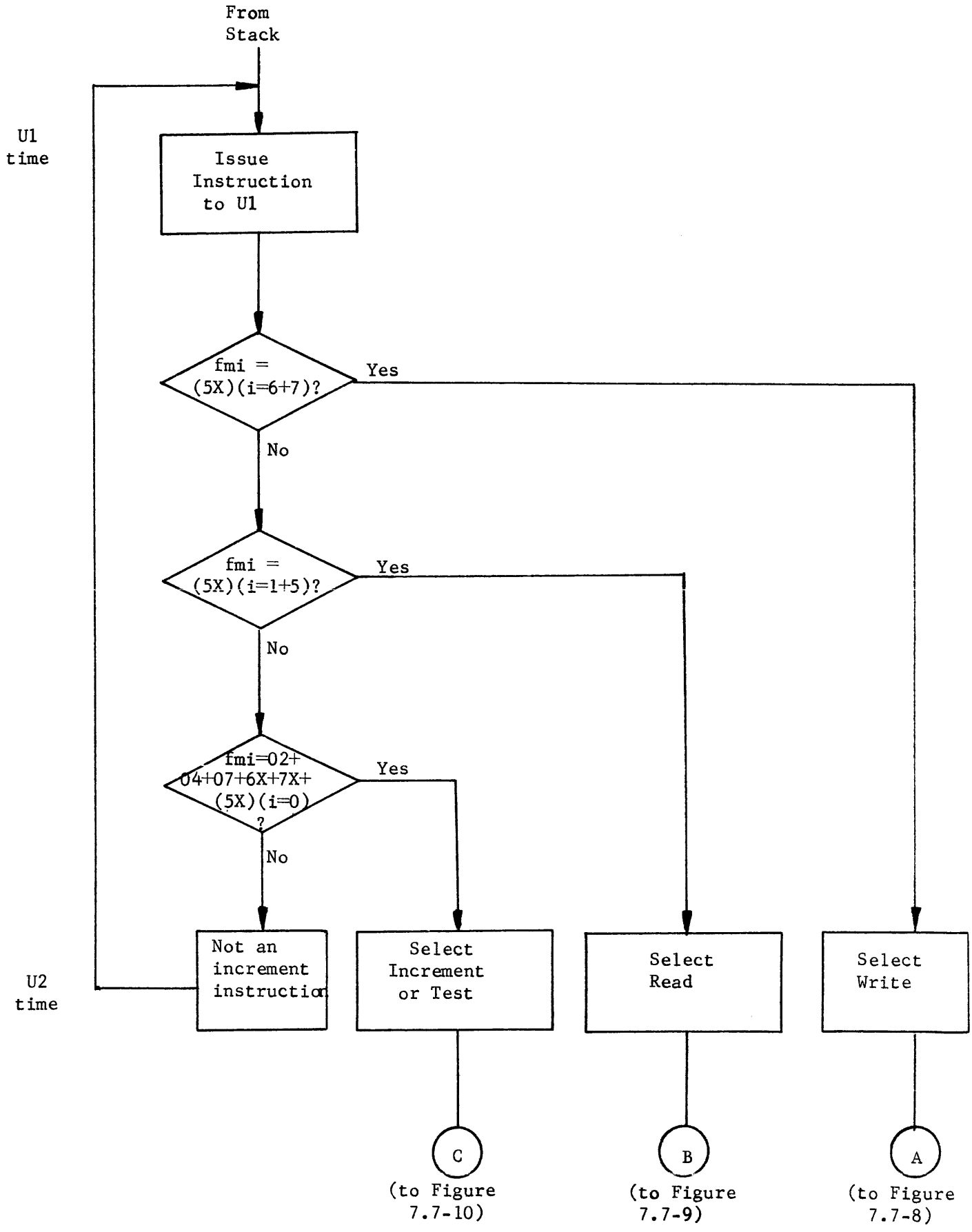
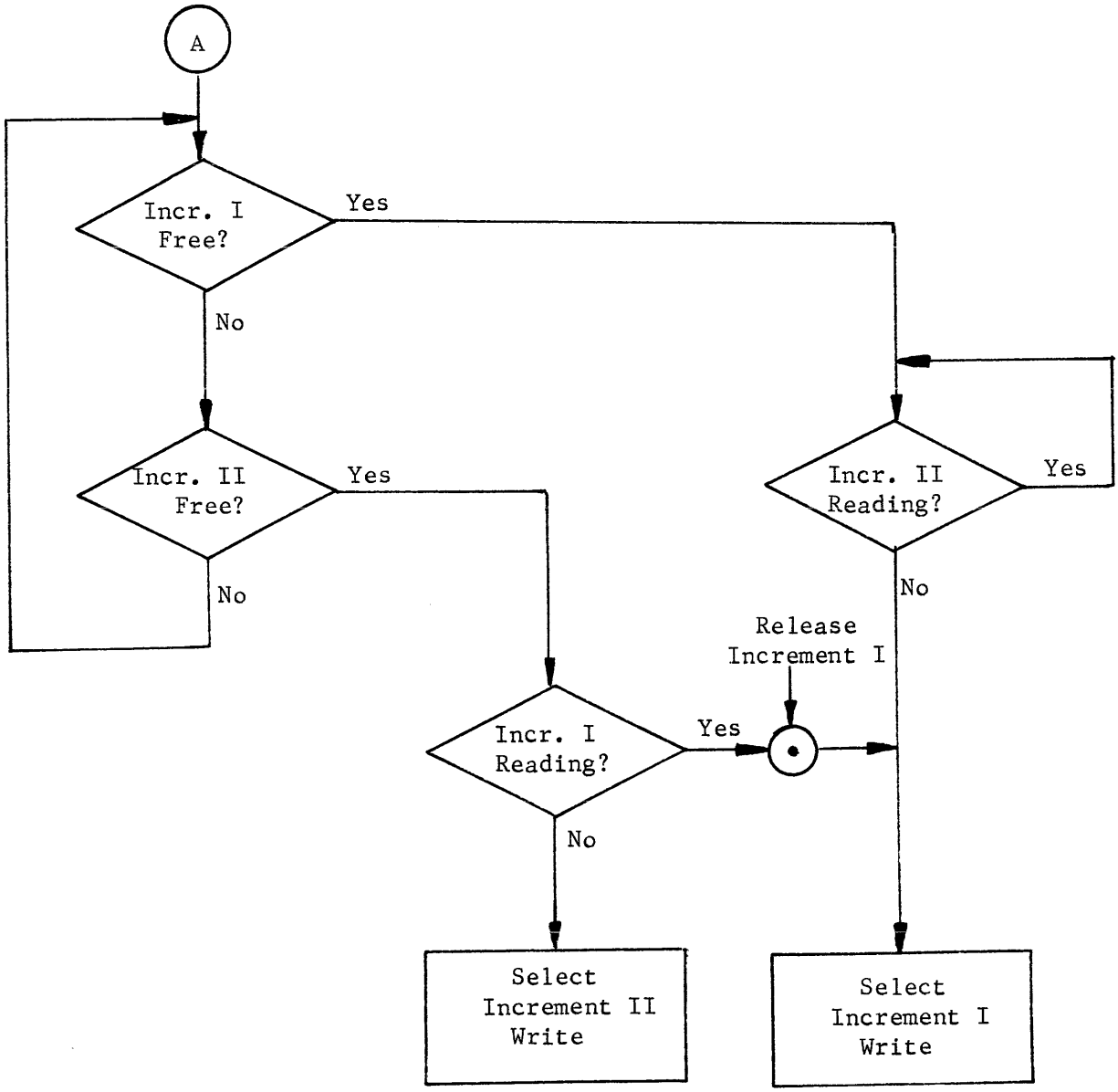
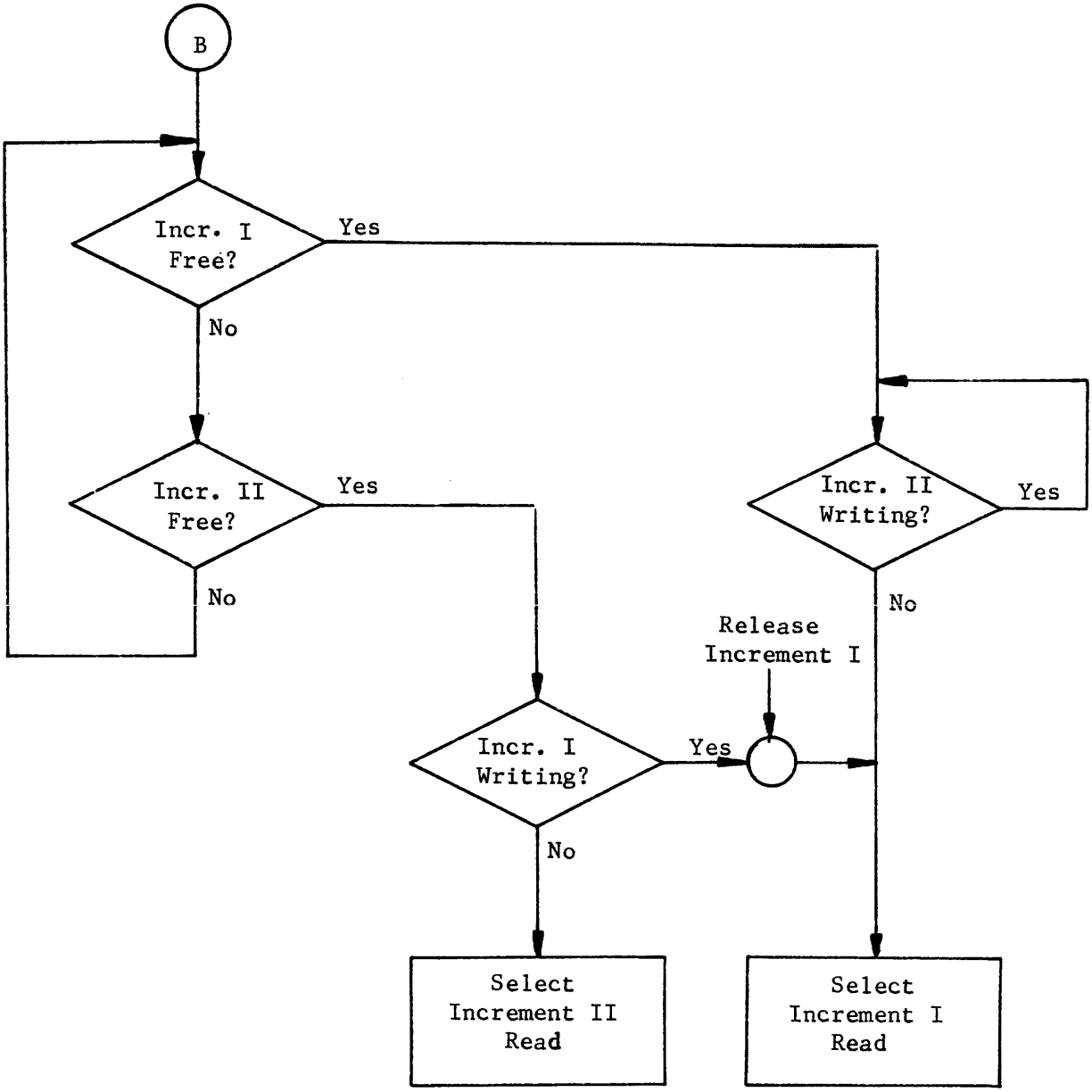

From Stack
U1 time
Issue Instruction to U1
fmi = (5X)(i=6+7)?
Yes
No
fmi = (5X)(i=1+5)?
Yes
No
fmi=02+ 04+07+6X+7X+ (5X)(i=0) ?
Yes
No
U2 time
Not an increment instruction
Select Increment or Test
Select Read
Select Write
C
(to Figure 7.7-10)
B
(to Figure 7.7-9)
A
(to Figure 7.7-8)

Figure 7.7-7



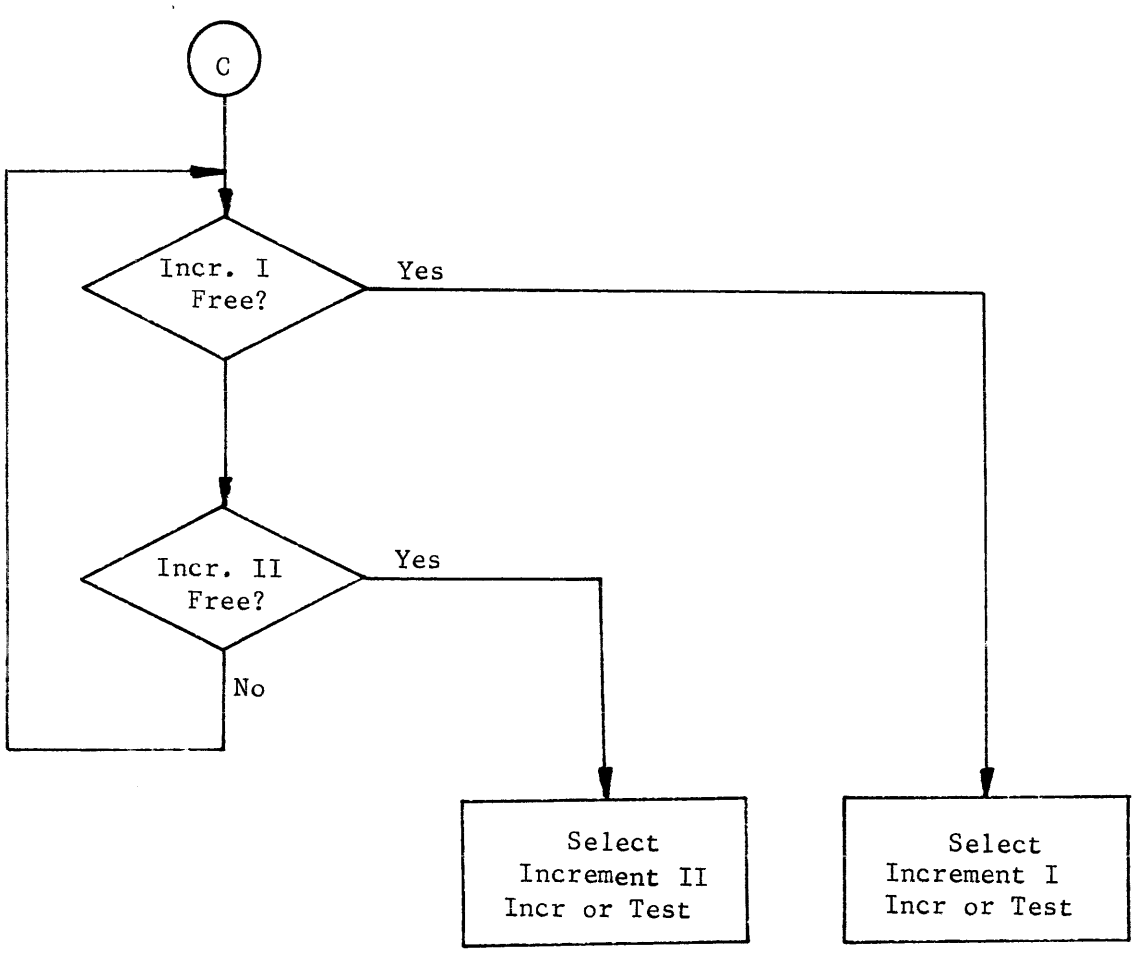
Select Increment Write

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 Unit 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":

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

---

*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

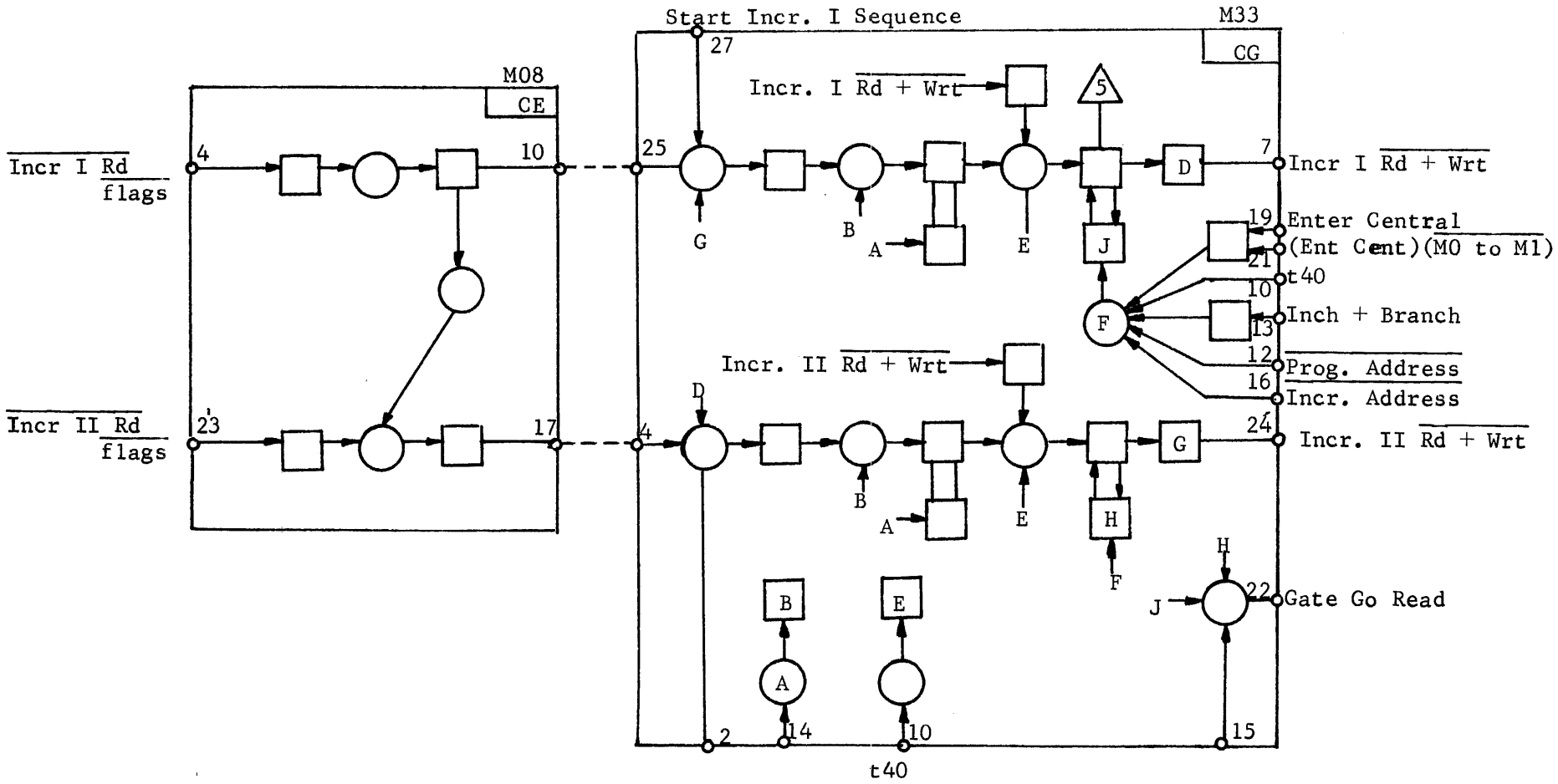


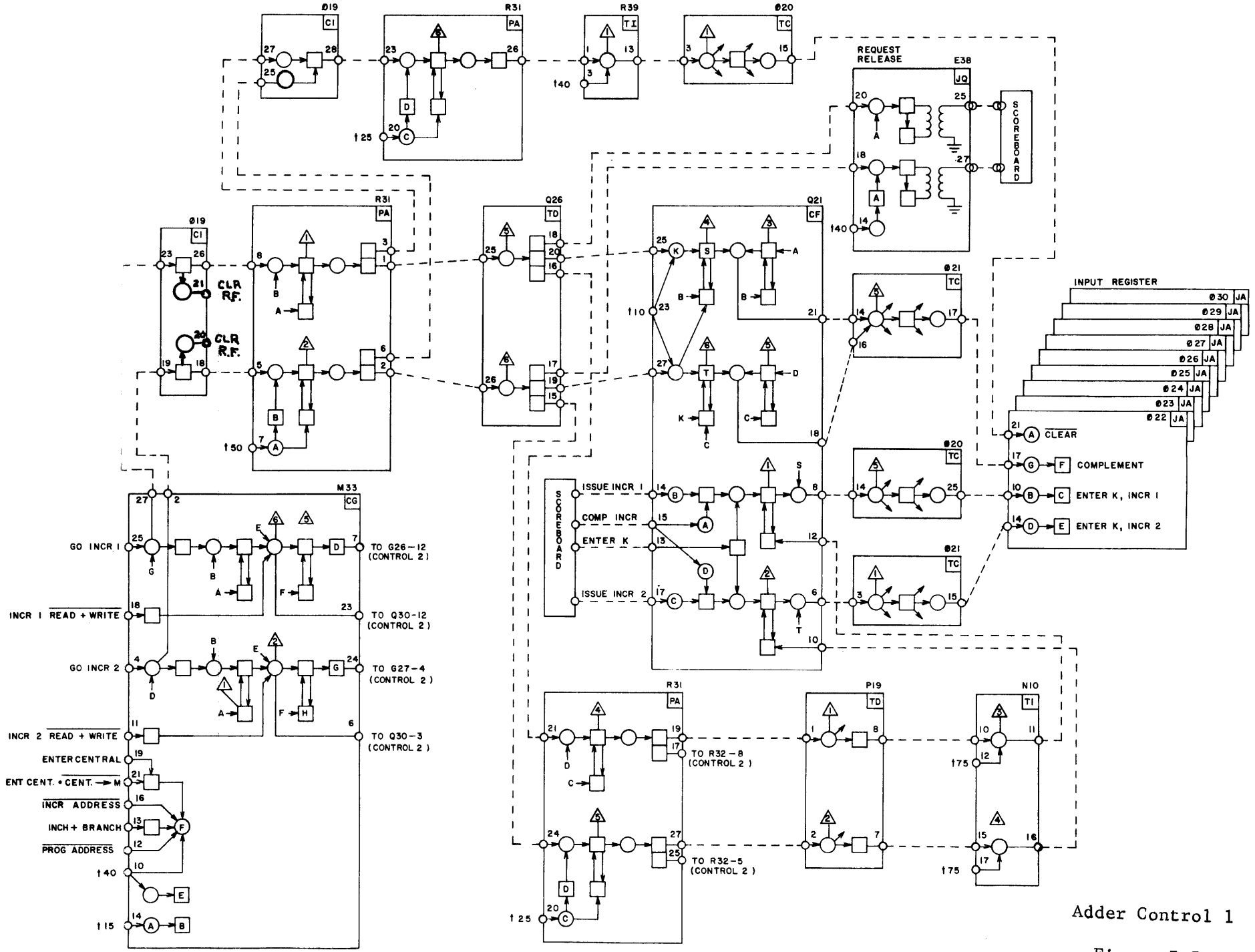
Figure 7.7-11

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:

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

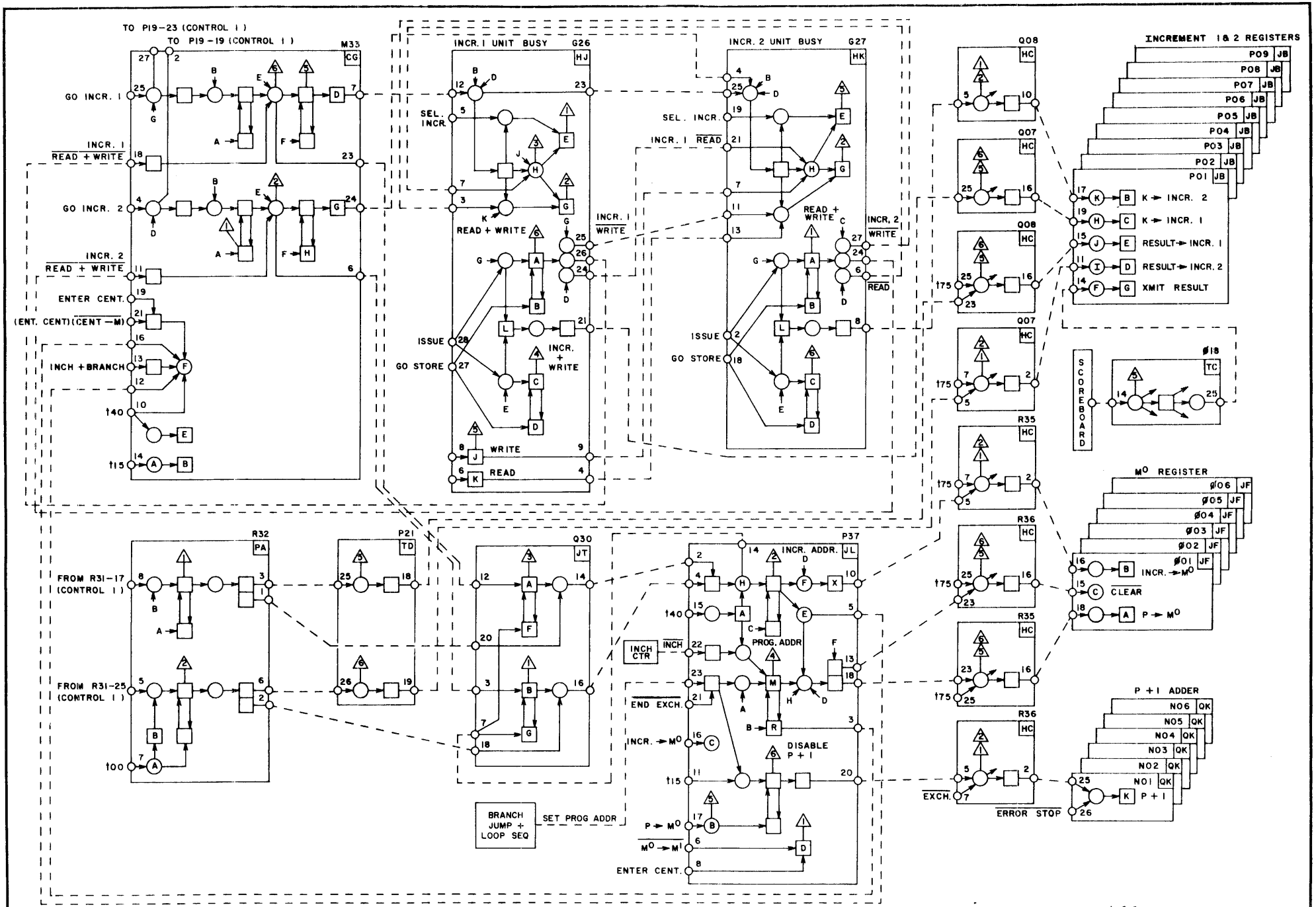
When priority is finally granted for the RNI address, the "M0 to M1" gate will be generated. This then will enable "Incr to M0" which in turn clears the Increment Address flip/flop. "M0 to M1" will also clear the Enter Central flip/flop. All conditions required by term "F" are now met. M33, TP5 will 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:

Borrow = 0 / 0

Satisfy = 1 / 1

Enable = 1 / 0 or 0 / 1

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:

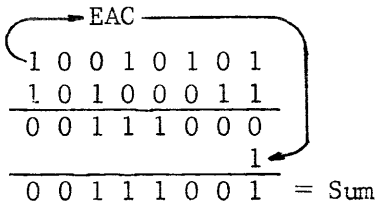
True Operands (if adding to add)	Second Operand Complemented (if subtracting to add)	Condition of Stage
0 0	0 1	Borrow
1 1	1 0	Satisfy
0 1	0 0	Enable
1 0	1 1	Enable

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.



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:

S B E E B E E S	
1 0 0 1 0 1 0 1	operand #1
1 0 1 0 0 0 1 1	operand #2
0 0 1 1 0 1 1 0	2. "exclusive OR"
1 1 1 1 0 0 0 0	3. borrows into stage
0 0 1 1 1 0 0 1	4. "Equivalence"

1. label stages

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:

17 15	14 12	11 9	8 6	5 3	2 0
Group --- 5	4	3	2	1	0

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 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 $\overline{k}$, and $\overline{k}$. 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 <u>not</u> contain an enable (Not "exclusive OR")
2. Check for a borrow leaving each stage. Using Stage 2^0 as an example,

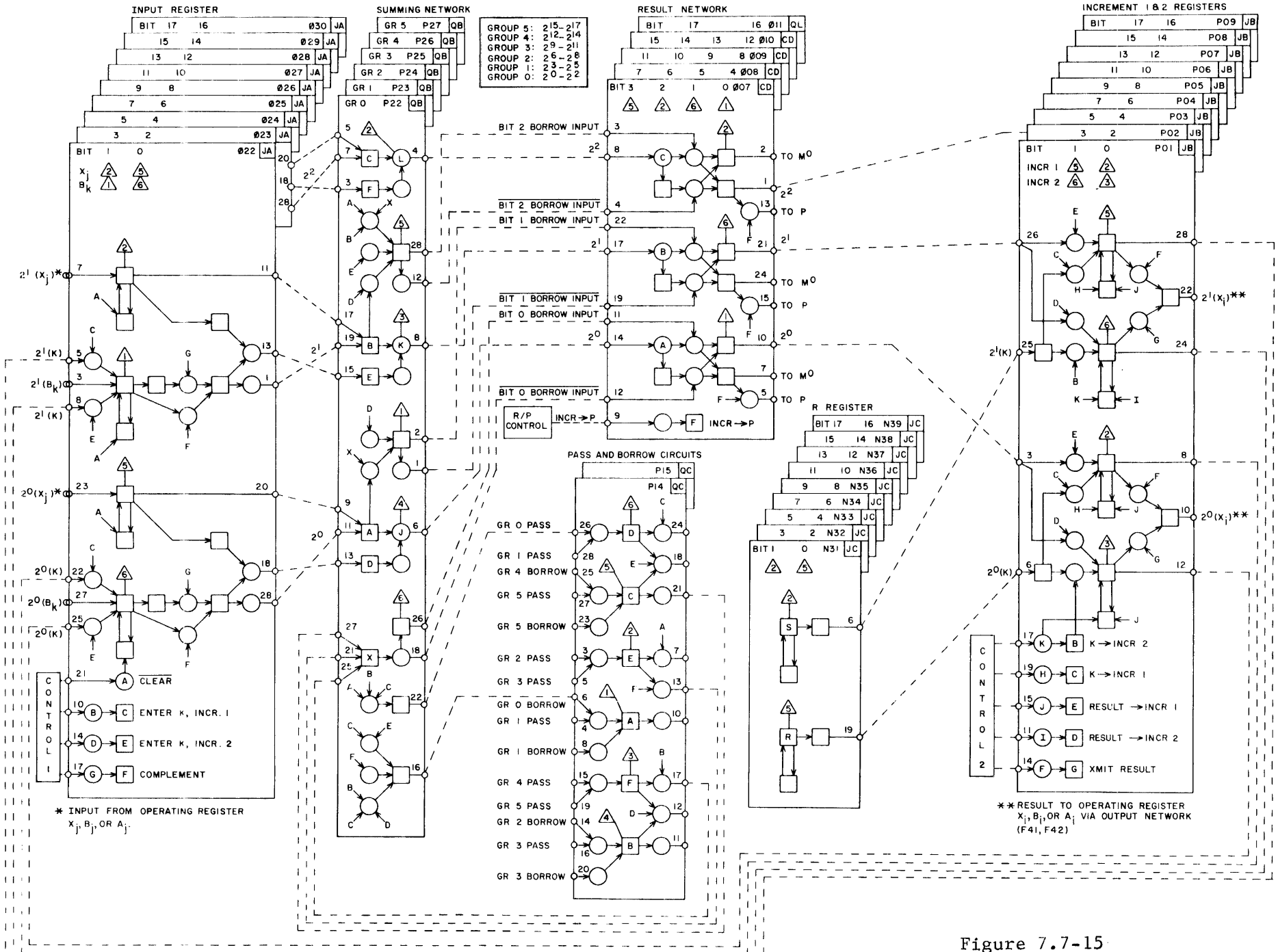

INPUT REGISTER
SUMMING NETWORK
RESULT NETWORK
INCREMENT 1 & 2 REGISTERS
GROUP 5: 2^15 - 2^17
GROUP 4: 2^12 - 2^14
GROUP 3: 2^9 - 2^11
GROUP 2: 2^6 - 2^8
GROUP 1: 2^3 - 2^5
GROUP 0: 2^0 - 2^2
BIT 2 BORROW INPUT
BIT 1 BORROW INPUT
BIT 0 BORROW INPUT
R/P CONTROL
INCR → P
R REGISTER
PASS AND BORROW CIRCUITS
GR 0 PASS
GR 1 PASS
GR 4 BORROW
GR 5 PASS
GR 5 BORROW
GR 2 PASS
GR 3 PASS
GR 0 BORROW
GR 1 PASS
GR 1 BORROW
GR 4 PASS
GR 5 PASS
GR 2 BORROW
GR 3 PASS
GR 3 BORROW
CONTROL 1
CLEAR
ENTER K, INCR. 1
ENTER K, INCR. 2
COMPLEMENT
CONTROL 2
K → INCR 2
K → INCR 1
RESULT → INCR 1
RESULT → INCR 2
XMIT RESULT
* INPUT FROM OPERATING REGISTER
Xj, Bj, OR Aj
** RESULT TO OPERATING REGISTER
Xi, Bi, OR Ai VIA OUTPUT NETWORK
(F41, F42)

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
$(\overline{\text{Borrow}})$(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:

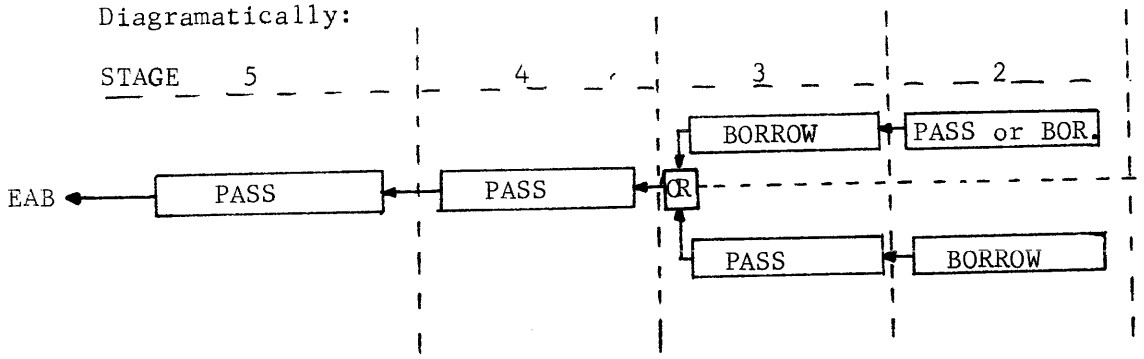
$$\text{Pin } 17 = 0 \Leftrightarrow F \cdot B$$

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

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

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

Diagramatically:



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	Bi = Bj
	05	Bi ≠ Bj
Magnitude	06	Bi ≥ Bj
	07	Bi < Bj

Although both tests use the adder logic, only the magnitude test checks the result of the complete add process. Since Bk (Bj of opcode) is complemented during Branch tests, the adder subtracts Bk from Bj. 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 (Bk) is in complement form, equality can be determined by looking for

an exclusive OR" in each stage. For example,

if feeders equal:	if the true operands equal:
1 or 0	1 or 0
0 1	1 0
(exclusive OR)	(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 Ø07, 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":

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

or

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

Magnitude Test

Two tests, which take place simultaneously are used to determine whether $Bi \geq Bj$ or $Bi < Bj$. (Figure 7.7-16).

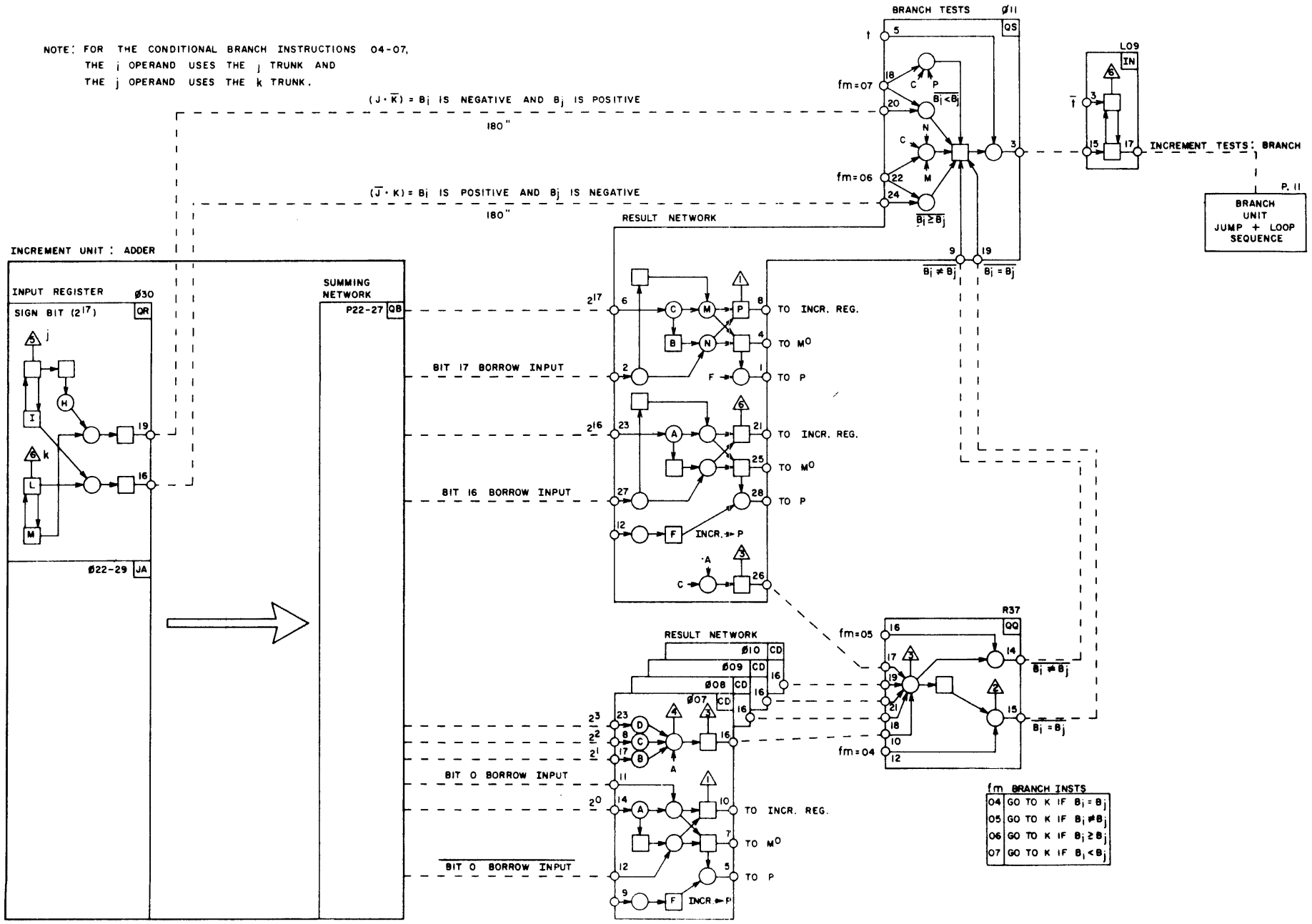

NOTE: FOR THE CONDITIONAL BRANCH INSTRUCTIONS 04-07,
THE i OPERAND USES THE j TRUNK AND
THE j OPERAND USES THE k TRUNK.
(J · K̄) = Bi IS NEGATIVE AND Bj IS POSITIVE
180"
(J̄ · K) = Bi IS POSITIVE AND Bj IS NEGATIVE
180"
BRANCH TESTS Ø11 QS
fm=07
fm=06
Bi<Bj
Bi≥Bj
LO9 IN
INCREMENT TESTS: BRANCH
P. 11
BRANCH UNIT JUMP + LOOP SEQUENCE
RESULT NETWORK
INCREMENT UNIT : ADDER
INPUT REGISTER Ø30
SIGN BIT (2^17) QR
Ø22-29 JA
SUMMING NETWORK
P22-27 QB
2^17
BIT 17 BORROW INPUT
2^16
BIT 16 BORROW INPUT
TO INCR. REG.
TO M^0
TO P
INCR.→P
RESULT NETWORK
Ø10 CD
Ø09 CD
Ø08 CD
Ø07 CD
2^3
2^2
2^1
BIT 0 BORROW INPUT
2^0
BIT 0 BORROW INPUT
R37 QQ
fm=05
fm=04
fm BRANCH INSTS
04 GO TO K IF Bi = Bj
05 GO TO K IF Bi ≠ Bj
06 GO TO K IF Bi ≥ Bj
07 GO TO K IF Bi < Bj

Figure 7.7-16

1) The signs (bit 17) of the two operands are compared. If they are different, the positive number is recognized as the larger of the two. The sign comparison is made on Ø30, pins 19 and 16, and is sent to module Ø11 where the operand signs are ANDed with the Branch op code (06 or 07). For example, if Bi is negative and Bj is positive (Ø11, pin 20 = 1) and fm = 07 (pin 18 = 1), the branch condition is met since Bi $\prec$ Bj. If Bi is positive and Bj is negative (Ø11, pin 24 = 1) and fm = 06 (pin 22 = 1) Bi $\succeq$ Bj and again the branch condition is met (L09, pin 17).

2) If the operand signs are alike, the one's complement difference (Bi − Bj) is taken and a check for a Borrow into stage 17 is made. A borrow indicate that Bi $\preceq$Bj. This condition, given by terms C and P(which indicate equivalence in bits 17 and a borrow in) is ANDed with fm = 07 (Ø11, pin 18) to generate a "condition met" signal (L09, pin 17). If fm = 06, the signs are alike, and no Borrow is propagated to stage 17, the condition is again met since Bi $\succeq$ Bj for this case. This is checked on Ø11, pin 22, AND terms N, M and C.

Module Ø11, pin 3 = "0" clears L09, TP6. This occurs through ORing the possible condition met gates on Ø11.

1) (fm = 04)(Equivalence)
2) (fm = 05)($\overline{\text{Equivalence}}$)
3) (fm = 06)(Bi $\succeq$ Bj)
4) (fm = 07)(Bi $\prec$ Bj)

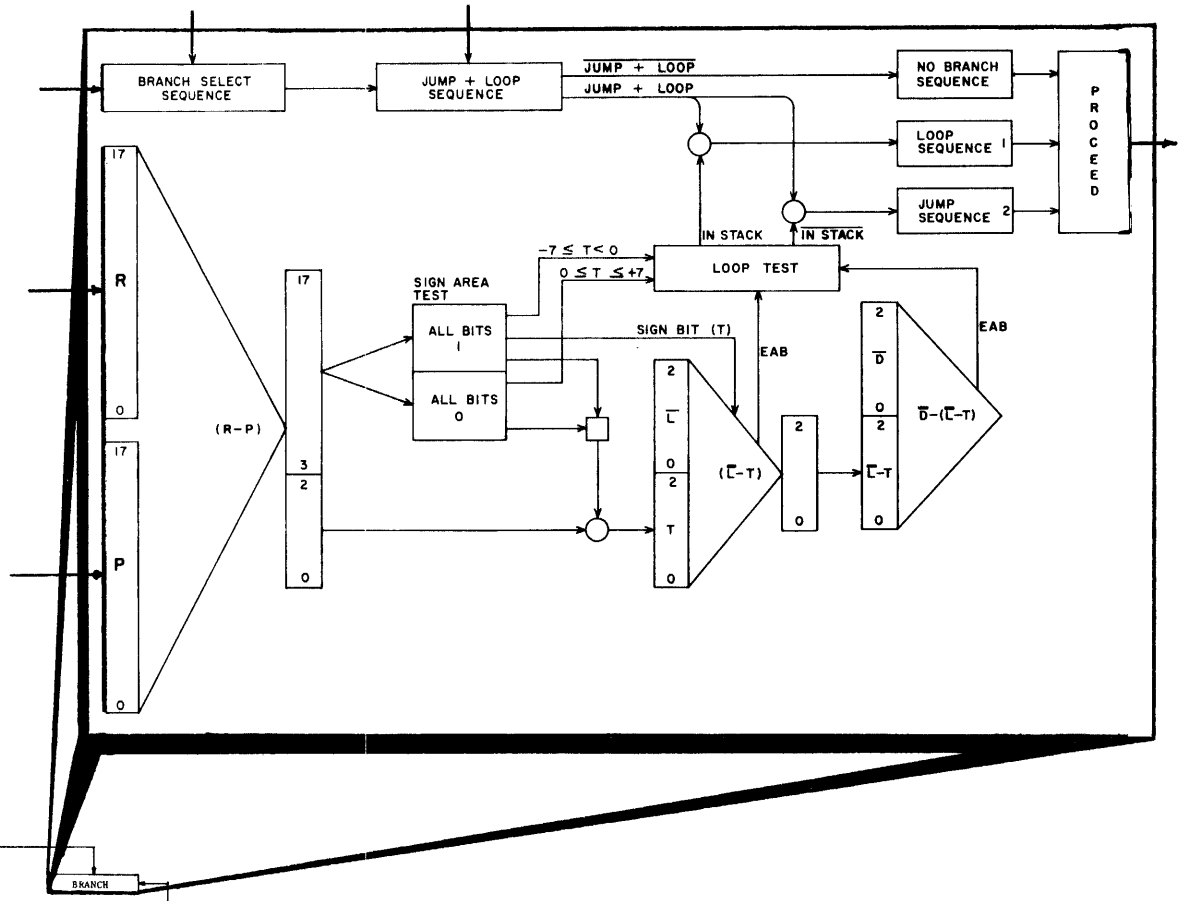
The clear side of L09, TP6 feeds pin 17 which sends the "Condition Met" signal to the Branch unit.

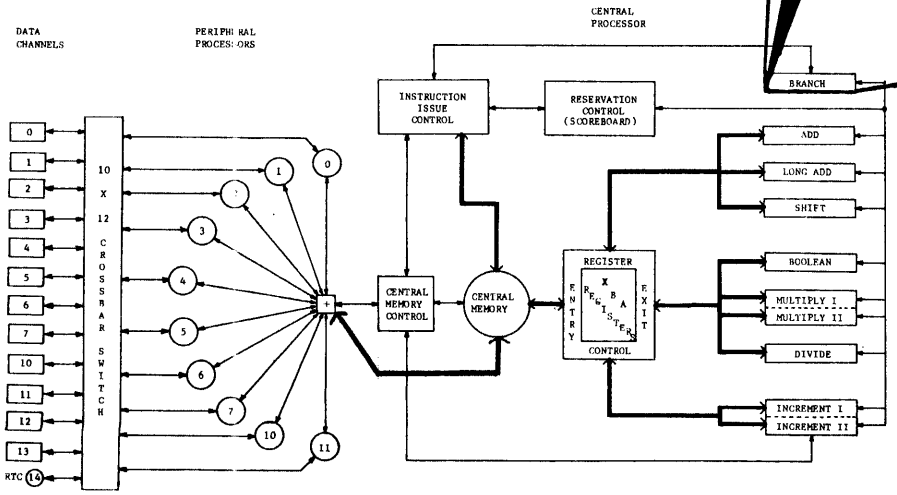
SECTION 7.8

BRANCH

FUNCTIONAL UNIT

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 Bi + K

2. Conditional Branches

   03

   04 - 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). When the new instruction word is read from memory issuance of the instructions begins with parcel zero of IO.

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. Results of the tests

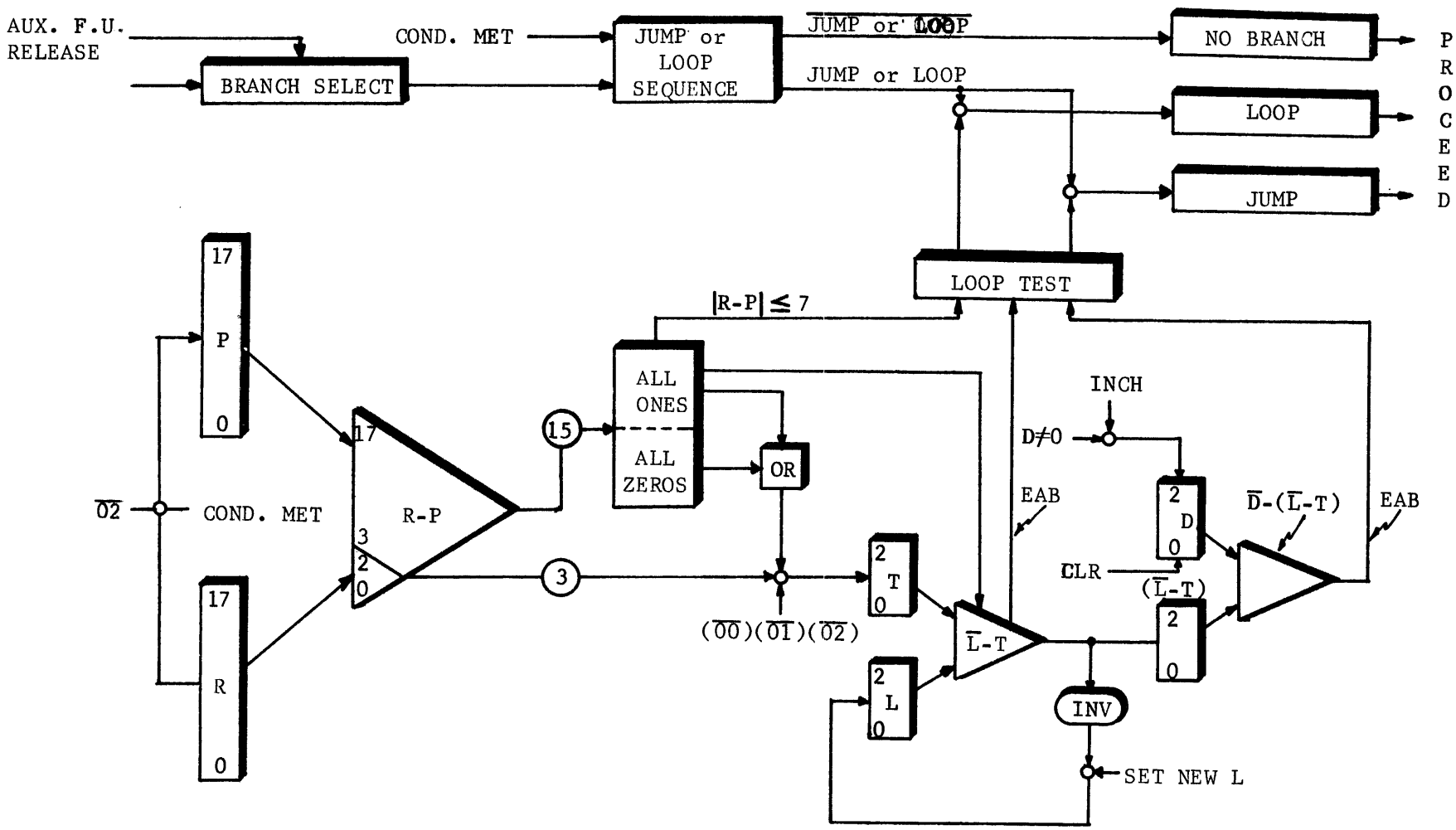


FIGURE 7.8-1

BRANCH FUNCTIONAL UNIT

are sent to the Branch unit where they are logically combined (ANDed) with a translation of the opcode being processed. Either the condition is met or it is not.

2. In conjunction with the "Condition Met" test (paragraph #1) the "in Stack/Out Stack" tests are performed. These are made in the Branch unit and will determine if a branch can be made in the stack or whether a memory reference must be made to the jump address. 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 register (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 I0, 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 issue begin the new "in stack" program. (Refer to Section 7.8.7 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 M^{o} 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 I^{o} will be the first instruction issued. (Refer to Section 7.8.8 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 M^0
6. Send S to Memory write distributor and force 0400 into write distributor.
7. Increment P (Jump Address plus 1) and send to M^0
8. Send M^0 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 M^0 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 K if Xj = 0	(ZR Xj K)
031	Jump to K if Xj $\neq$ 0	(NZ Xj K)
032	Jump to K if Xj is Positive	(PL Xj K)
033	Jump to K if Xj is Negative	(NG Xj K)
034	Jump to K if Xj is In range	(IR Xj K)
035	Jump to K if Xj is Out of range	(OR Xj K)
036	Jump to K if Xj is Definite	(DF Xj K)
037	Jump to K if Xj 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 point 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 Xj 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 Xj operand from Register Exit control and performs the four tests (zero, sign, infinite, and indefinite) simultaneously. Four signals may result from testing of Xj 1) Xj $\neq$ 0, 2) Xj $<$ 0, 3) Xj = out of range, or 4) Xj = 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:

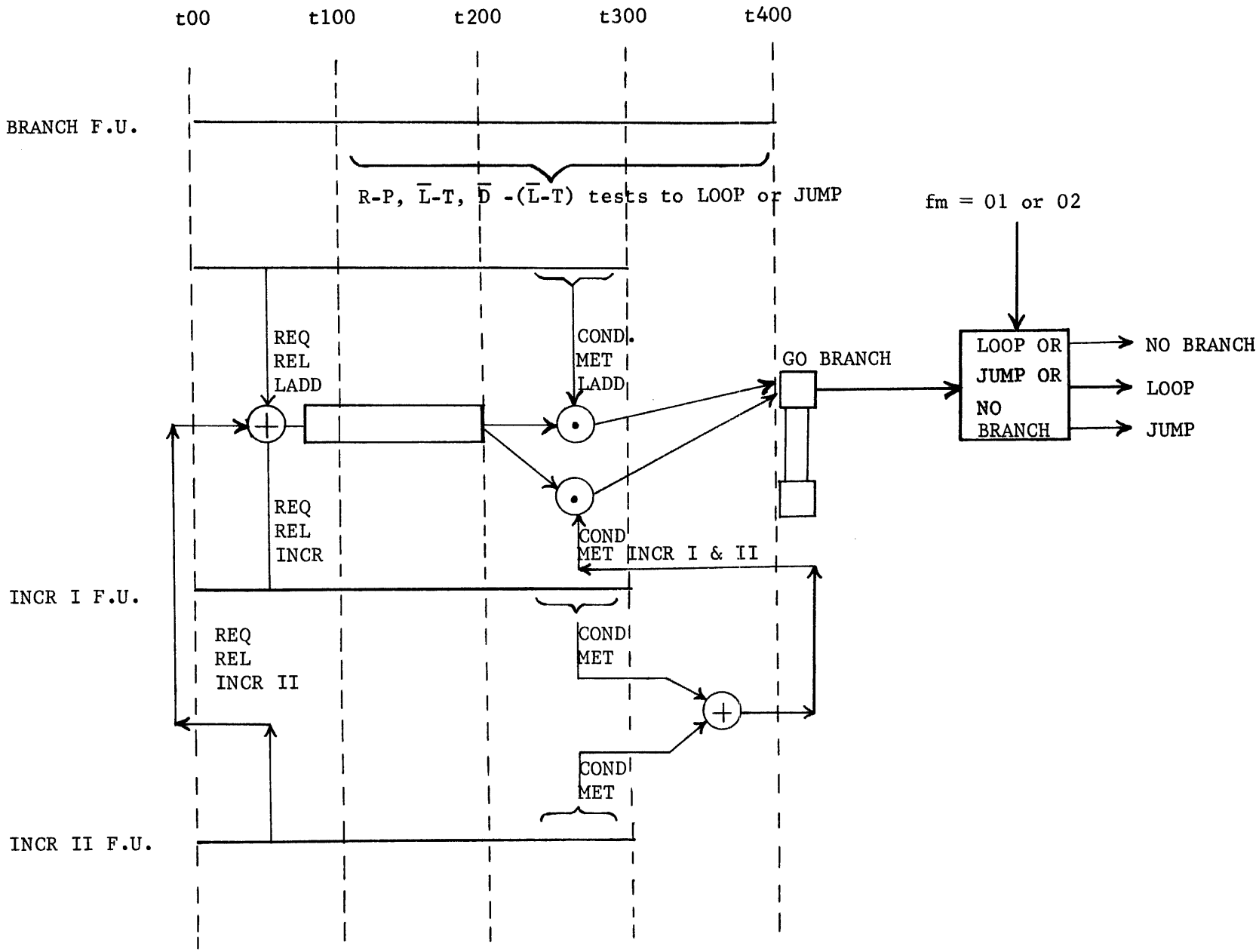

t00
t100
t200
t300
t400
BRANCH F.U.
R-P, $\overline{L}$-T, $\overline{D}$ -($\overline{L}$-T) tests to LOOP or JUMP
fm = 01 or 02
REQ REL LADD
COND. MET LADD
GO BRANCH
LOOP OR JUMP OR NO BRANCH
NO BRANCH
LOOP
JUMP
REQ REL INCR
COND MET INCR I & II
INCR I F.U.
REQ REL INCR II
COND MET
COND MET
INCR II F.U.

Figure 7.8-2

(Condition Not Met) • (Loop + Jump) $\Longrightarrow$ No Branch

(Condition Met) • (Loop) $\Longrightarrow$ Branch In Stack

(Condition Met) • (Jump) $\Longrightarrow$ 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 (0X) 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 I0.

040 Jump to K if Bi = Bj (EQ Bi Bj K)

050 Jump to K if Bi $\neq$ Bj (NE Bi Bj K)

060 Jump to K if Bi $\geq$ Bj (GE Bi Bj K)

070 Jump to K if Bi $<$ Bj (LT Bi Bj K)

Definition:

These instructions test the 18-bit word in Bi against the 18-bit word in Bj (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 (Bi = Bj, Bi $\neq$ Bj, Bi $\geq$ Bj, and Bi $<$ Bj) 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 circuity as was explained for the 03X branch instructions. Reference is therefore made to the sequence discussion of the 03X instructions for further explanation of the No Branch, Loop, and Jump cases.

7.8.3 TIMING SEQUENCE

Five general cases of Branch functional unit timing are discussed in this section. In all five cases, the time zero reference on the timing charts is the Scoreboard Issue of the particular Branch case. The five cases cases discussed are:

1) No Branch
2) Loop
3) Jump
4) Return Jump
5) Exit Mode Stop

In following the timing discussions and corresponding timing charts, the C. E. Diagrams and Chassis #5 wire tabs should be referenced.

NO BRANCH SEQUENCE - See Figure 7.8-3.1

1. F38 - TP2 Branch Select F/F Set by (U_2 Fm = ox)(Scbd issue)

2. G30 - TP1 Go F/F Cleared by (U_2 Fm=ox)(Scbd issue)

3. L21 - 13 Auxiliary Functional Unit release signal. This could be from L20-13 or L22-13, depending on the functional unit used. It indicates that the functional unit has timed out and its' results are ready.

4. R31 -TP3, R30-TP3 and R30 - TP6 These are stages ofthe Branch unit timing chain which enalde setting the stack controls (if necessary) and proceeding in the proper sequence.

5. Q04 - 21 This signal is generated to set the no branch sequence if the Jump or Loop F/F was not set (condition not met)

6. L14 - TP6 This is the first F/F of a section of this sequence which will allow resuming the instruction following the branch.

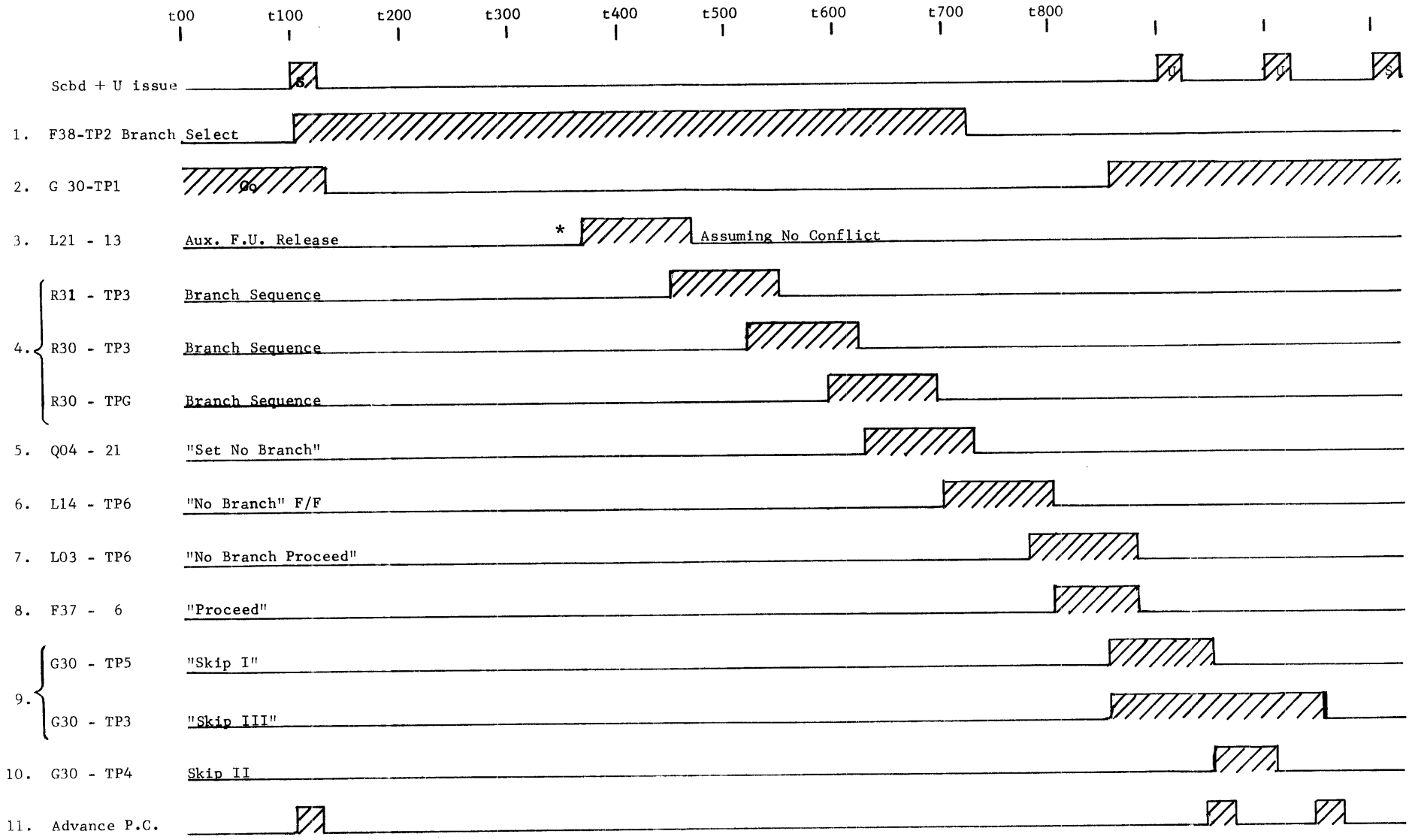
7. L03 - TP6 The proceed resulting from the no branch sequence occurs at this time.

8. F37 - G "Proceed" signal to set "Go", "Skip I" and "Skip III"

9. G30 TP1, TP3 and TP5 The "Go", "Skip I" and "Skip III" F/Fs are set at this time in preparation for issuing the next parcel.

10. G30-TP4 "Skip II" F/F is set to disable clearing "Skip III" for another minor cycle.

11. Advance Parcel Counter This signal is generated at the beginning of the branch sequence by (U_2fm = ox)(Scbd issue) and at the end of the sequence by (Proceed)(issue)($\overline{U_1 fm = OX}$)(t25).



BRANCH UNIT TIMING - NO BRANCH

* Increment Release at t375
Long Add Release at t475

Figure 7.8-3.1

LOOP SEQUENCE - See Figure 7.8-3.2

1.) Begin with a "SCBD Issue" into F38/3 (sheet 100) at the time t90 previous to time too of this graph.

1A.) K⟶R transfer is done on every "Issue", whether or not the content of R is used. (M28 sheet 54)

2.) The "Branch Select" FF (F38-TP2 sheet 100) is set as a result of a (SCBD Issue)(FM=OK). It is cleared by the gate feeding Qo4/16 sheet 101.

3.) "Stop" is set a result of (OX)(SCBD Issue). See G36-TP3 on sheet 57.

4.) "Go" (G30-TP1 sheet 219) set by (Proceed)(t60) and cleared by Stop (3.above). "Proceed" defined in (12) below.

5.) G26-TP4 sheet 56 set while Incr. I is performing tests to see if condition mct.

5A.) L21-TP6 sheet 100 is set by (Issue)$(\overline{\text{error}})$ and cleared by a functional unit reservation code of XXX1.

6.) The Auxiliary Functional Unit Release enables the continuation of the branch sequence. (H25/16) sheet 100.

7.) After receiving a "Release" from the Aux. F.U., the Aux. F.U. timing chain (sheet 101 of F.U. prints) is started. The "condition met" signal from the Aux. F.U. and a time t50 from timing chain seats the "Jump + Loop" FF (R37-TP4 sheet 101). The remainder of the sequence is not timed, although, some consideration must be given to wire lengths.

8.) Go Branch is a logical 1 when (Inch) + (Jump + Loop)is present. Therefore "Go Branch"⟹$(\overline{\text{Inch}})$(Jump + Loop)

9.) "Disable Adv. P" (Q04/26 sheet 101) is a result "Jump + Loop" FF being set.

10.) R⟶P is enabled by P10/1 sheet 101 which translates as $[(\overline{\text{RJ+EM}})(\text{P} \quad \text{M}^{\circ})(\overline{\text{FM = 02}})]\ [(\overline{\text{INCH}})(\text{Jump + Loop})]$

11.) To clear PC, F33/23 sheet 102 must translate as [(Go Branch + RJ + EM] [in stack] . This is a function of R37-TP4 since Go Branch (Jump + Loop)$(\overline{\text{Inch}})$.

12.) "Proceed" entering G30/11 sheet 219 is a direct result of the "loop proceed" generated at F33/21 sheet 102 by [Go Branch + RJ + EM] [in stack] . See (11.) above.

13.) Skip I (G30-TP5 sheet 219) is set by the "Proceed" defined in (12) above.

14.) The setting of Skip I will result in setting skip II (G30-TP4 sheet 219)

15.) Skip III (G30-TP3 sheet 219) is set for 200ns as a result of the "Proceed" and setting skip II.

L O O P

1.	F34 ⟶ F38/3	SCBD Issue U issues are fanned out from F34 also.
1A.	K ⟶ R.	M28 sheet 54
2.	F38 - TP2	Branch Select
3.	G36 - TP3	Stop
4.	G30 - TP1	Go
5.	G26 - TP4	Incr. I Unit Busy
5A.	L21 - TP6	Select Release Incr. I
6.	H25/16	Release
7.	R37 - TP4	Jump + Loop
8.	P10 - TP3	$\overline{\text{Go Branch}}$
9.	Q04/26	Disable Advance P
10.	N31/"C"	R ⟶ P
11.	G31/"C"	Clear PC
12.	G30/11	Proceed
13.	G30 - TP5	Skip I
14.	G30 - TP4	Skip II
15.	G30 - TP3	Skip III

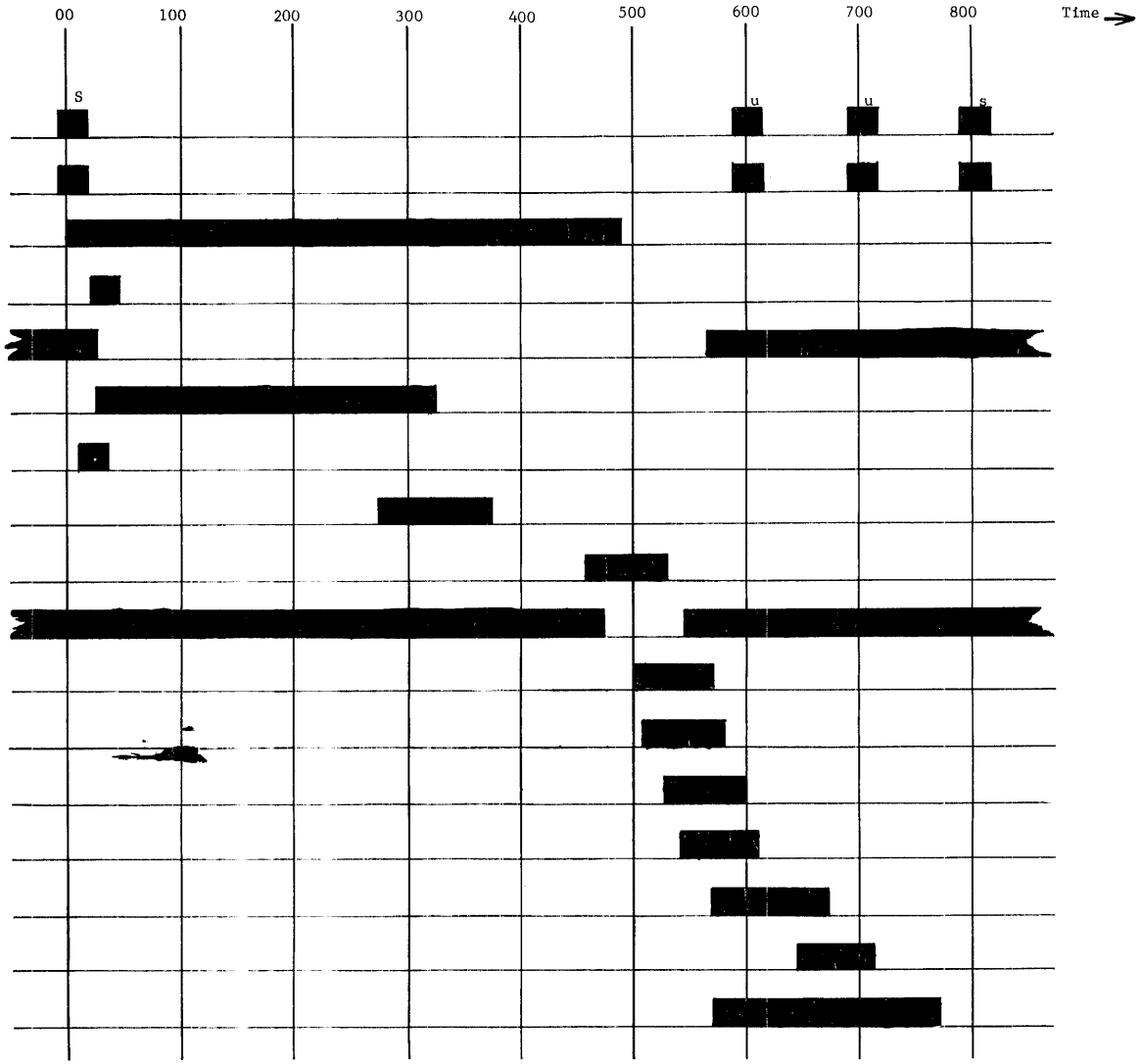


Figure 7.8-3.2

UNCONDITIONAL JUMP INSTRUCTION (Fm = 02) - See Figure 7.8-3.3

NOTE: The following duscussion assumes

No Functional Unit or Memory Bank Conflicts

1. Branch select FF is set by (Fu = OX)(SCBD issue)

2. Go is cleared by 'stop' via (OX)(SCBD issue)(page 24.1) & stays cleared until "proceed" sets it.

3. Aux. F.U. is issued 25 ns after branch select FF sets.

4. Aux. F.U. is released (Q = XX1X)(release INCR2) page 9 (H25 TP5).

5. Jump + loop is set by aux. F.U. release, t60, & fm = 02.

6. $\overline{\text{Set no branch}}$ signal comes up when jump + loop is set, aux. F.U. & a t30. This condition is enabled when a zero is present on p.n 21, Q04.

7. FFA is set by pause, jump, or end exchange; in this particular case, it sets jump, zero on pin 21, F37. Stop C.P. is set by the setting of FFA anded with a t50, when stop C.P. sets it forces L=7, PC=0 and voids the instruction stack. A "one" signal on pin 28 of R32 will cause the "Enable Restart" flip flop to set on the next t25.

8. Program address sets via jump + loop FF at t450 & is gated with a t25 to enable p⟶MO at 595. The P⟶MO comes back and clears the Program Address at t605. Clearing Program Address knocks down P⟶MO at 695.

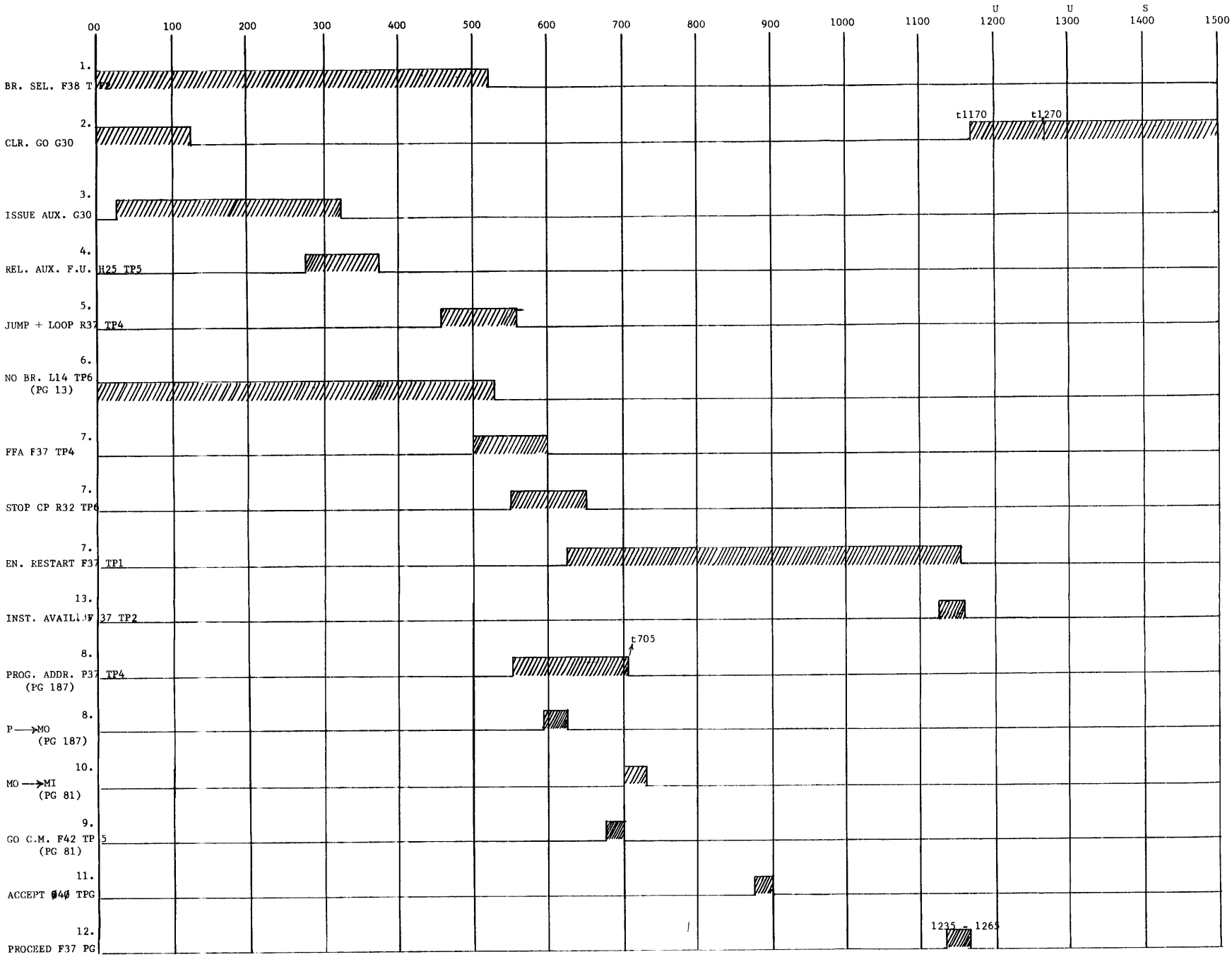
9. Go Central Memory comes up via (priority 2)(t75) page 81 which is at t675.

10. MO - always goes to M1 at a t00 time. This occurs at 700 on the timing chart for a jump instruction.

11. An "accept" signal (25NS) is returned from memory, 200 NS after sending the "go" to central memory. This occurs at t875 on the jump timing chart.

12. The "Accept" signal from central memory will generate the "proceed" signal (Pg. 23-F37 Pin 6) this allows the "Go" flip flop to set on the next T70.

13. Accept is anded with a tag 10 which is anded with Jump + Loop and sets the Instruction Available FF on the next t25.



JUMP TIMING CHART

Figure 7.8-3.3

RETURN JUMP SEQUENCE (fm = 01) - See Figure 7.8-3-4

1. Issues - Tooo is the scoreboard issue for the branch instruction. The u issue previous to this scoreboard issue enables our ∧ul ⟶ K transfer.

2. M1 ⟶ K - ∧ul register transferred to the K register with u issue.

3. Branch Select - Set by (SCBD issue)(Fm=OX) this will be cleared when Jump+ Loop Flip/Flop is set.

4. K ⟶ R - Jump Address sent to R Register. Occurs on the same SCBD issue which set branch select.

5. Gø F1,P/Flop - Cleared with (OX)(SCBD ISSUE) to stop issues during branch.

6. RJ + EM - Set at T75 by (Fm = 01)(SCBD Issue)($\overline{P \longrightarrow MO}$) It is cleared when TP 1, P10 is set, or when P ⟶ MO occurs.

7. P+1 ⟶ S - Unconditionally each T10. Return address gated to "s" Register.

8. S ⟶ Memory - Unconditionally each T50. Forms 0400(P+1) o——o at Jump address.

9. Disable P+1 - To allow true jump addres to be gated to MO from P.
   2nd Time - to allow true jump address + 1 to be gated to MO.

10. Program Address - Set to Enable our P ⟶ MO transfers.

11. R ⟶ P - Jump address gated to P register. Transfer occurs after RJ + EM Flip/Flop is set and Jump + Loop Flip/Flop is still clear.

12. P ⟶ MO - Jump address gated to MO. P+1 was disabled at this time.
    2nd P MO gates jump address + 1 to MO for our RNI.

13. ADV P - Jump address + 1 gated back to "P" register, which will be our address of our RNI.

14. PIO ⚠ - Set by P ⟶ MO. By setting this Flip/Flop we clear RJ + EM, set exit and then set Jump + Loop.

15. Jump + Loop - Set when exit Flip/Flop is set. This forces jump condition which clears branch select and sets FliP/Flop "A".

16. Exit Flip - Enables setting Jump + Loop Flip/Flop.

17. F37 Flip/Flop "A" - Set by a Jump condition which in turn enables setting stop CP Flip/Flop.

18. Stop CP Flip/Flop - Set when FF "A" is set to enable setting of "Enable" Restart Flip/Flop. Setting stop CP voids stack L = 7, D = 7, PC = 0.

19. Enable Restart - Set after stop CP FF is set to partially condition the the gate for the proceed after RNI.

20. MO ⟶ MI - Two of these occur. The first with a (tag 50) and the second with a (tag 10).

21. REQ RNI - Set after second MO ⟶ MI and our RNI accepted.

22. INST. AVAIL. - Set after Req RNI set to fully condition our gate for our proceed.

23. Proceed - Sets "Go" FF, Skip I FF, and Skip III FF to enable two u issues for the instruction at Jump Address + 1.

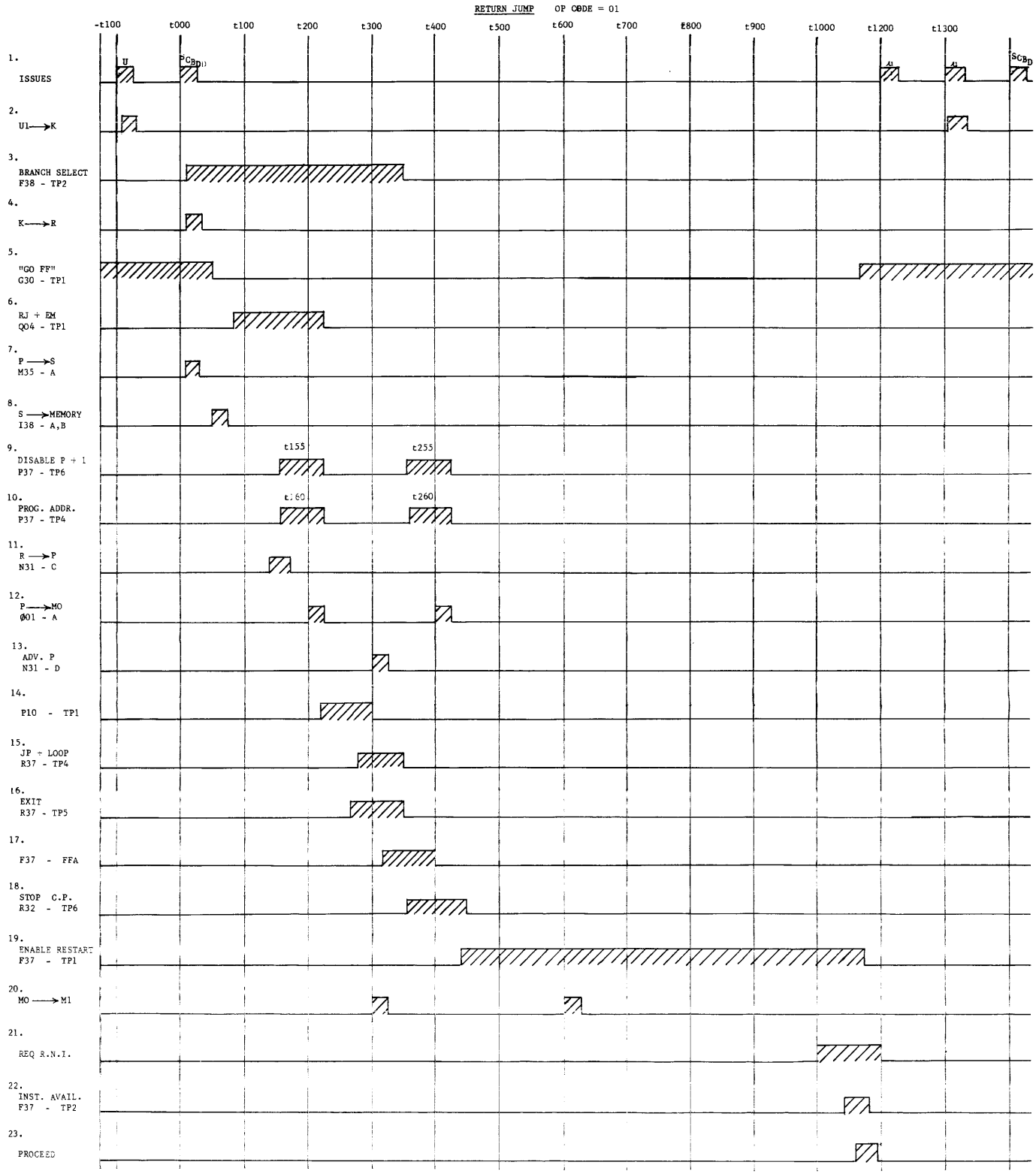

RETURN JUMP OP CODE = 01
-t100
t000
t100
t200
t300
t400
t500
t600
t700
t800
t900
t1000
t1200
t1300
1. ISSUES
2. U1 → K
3. BRANCH SELECT F38 - TP2
4. K → R
5. "GO FF" G30 - TP1
6. RJ + EM Q04 - TP1
7. P → S M35 - A
8. S → MEMORY I38 - A,B
9. DISABLE P + 1 P37 - TP6
t155
t255
10. PROG. ADDR. P37 - TP4
t160
t260
11. R → P N31 - C
12. P → MO Ø01 - A
13. ADV. P N31 - D
14. P10 - TP1
15. JP + LOOP R37 - TP4
16. EXIT R37 - TP5
17. F37 - FFA
18. STOP C.P. R32 - TP6
19. ENABLE RESTART F37 - TP1
20. MO → M1
21. REQ R.N.I.
22. INST. AVAIL. F37 - TP2
23. PROCEED

Figure 7.8-3.4

EXIT MODE STOP, RNI Out of Bounds and (EM) = xx1 - See Figure 7.8-3.5

1.) End Exch. sets F37 - TP4

2.) End Exch. also sets P37-TP4 (Program address FF)

3.) P37-TP4 enables P M° gates at Ø01 - A

4.) P37-R sets Enter Central FF (P37-TP3)

5.) F37-TP4 (Ref #1) sets stop C.P. (R32-TP6), which in turn sets Enable Restart (Ref #6), clears Parcel counter, and sets L+D=7.

6.) Set by R32-TP6 (Stop C.P.)(Ref #5)

7.) Parcel counter clears (Ref #5)

8.) RNI (R33-TP6) is set by a delayed accept (Ref #9) anded with Tag=(10).

9.) Accept (Ø40-TP6) from C.M. is delayed before setting RNI (R33-TP6) (Ref #8). This allows info to get to IR.

10.) RNI sets Inst. Available (F37-TP2) which together with Enable Restart (Ref #6) gives us a proceed which sets GO and Skip I and Skip III (Ref #11 & 12)

11.) Skip I (G30 - TP5) set by Proceed (Ref #10)

12.) Skip III (G30 - TP3) set by Proceed (Ref #10)

13.) Skip II (G30 - TP4) set after skip I

14.) Inst. Avail. (F37 - TP2) set by RNI (Ref #8)

15.) IR ⟶ IO (E25 - 10) initiated by RNI (Ref #8)

16.) Inch (G29-TP2) set by first issue with Pk=0 & L=7.

17.) Proceed (F37-6)(Ref #10)

18.) Error (F35 - TP4) set by M° FL (Ref. 21) and Skip I (Ref. #11)

19.) Big Inch (L19-TP6) set by Q26-7 (Ref #16)

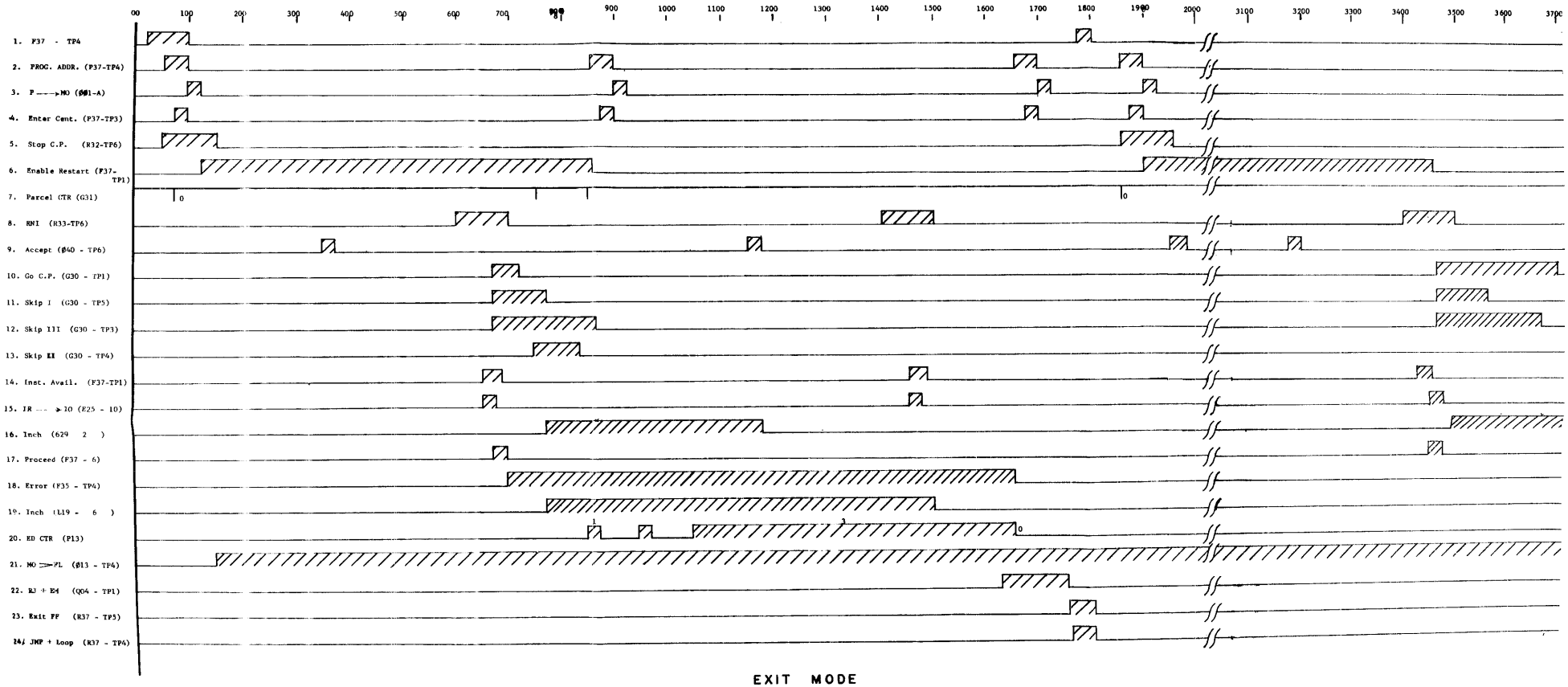
20.) EOK (P13) by Error (Ref #18)

21.) MO ⟹ FL is a static network that comes up when registers are set (Ref. #43)

22.) RJ + EM (Q04-TP1) set by Error Exit (EDK=3)($\overline{\text{Big Inch}}$) Ref #18 & 19

23.) Exit F.F. (R37-TP5) set by (RJ + EM)(P MO) Ref. #22 & #3

24.) Jump & loop (R37-TP4) Set by Exit (Ref. #23)



EXIT MODE

Figure 7.8-3.5

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 (03X or 04-07) 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) $\overline{L}$ - T test, and 3) $\overline{D}$ - ($\overline{L}$ - 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.

1) 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 physical stack registers.

Four cases may result from the R - P test:

a) 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.

b) 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.

c) 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 branched.

d) 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).

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. Figure 7.8-5 is a block diagram of the R-P test. For a bit-by-bit analysis of the logic, refer to sheet 96 in the C. E. Diagrams.

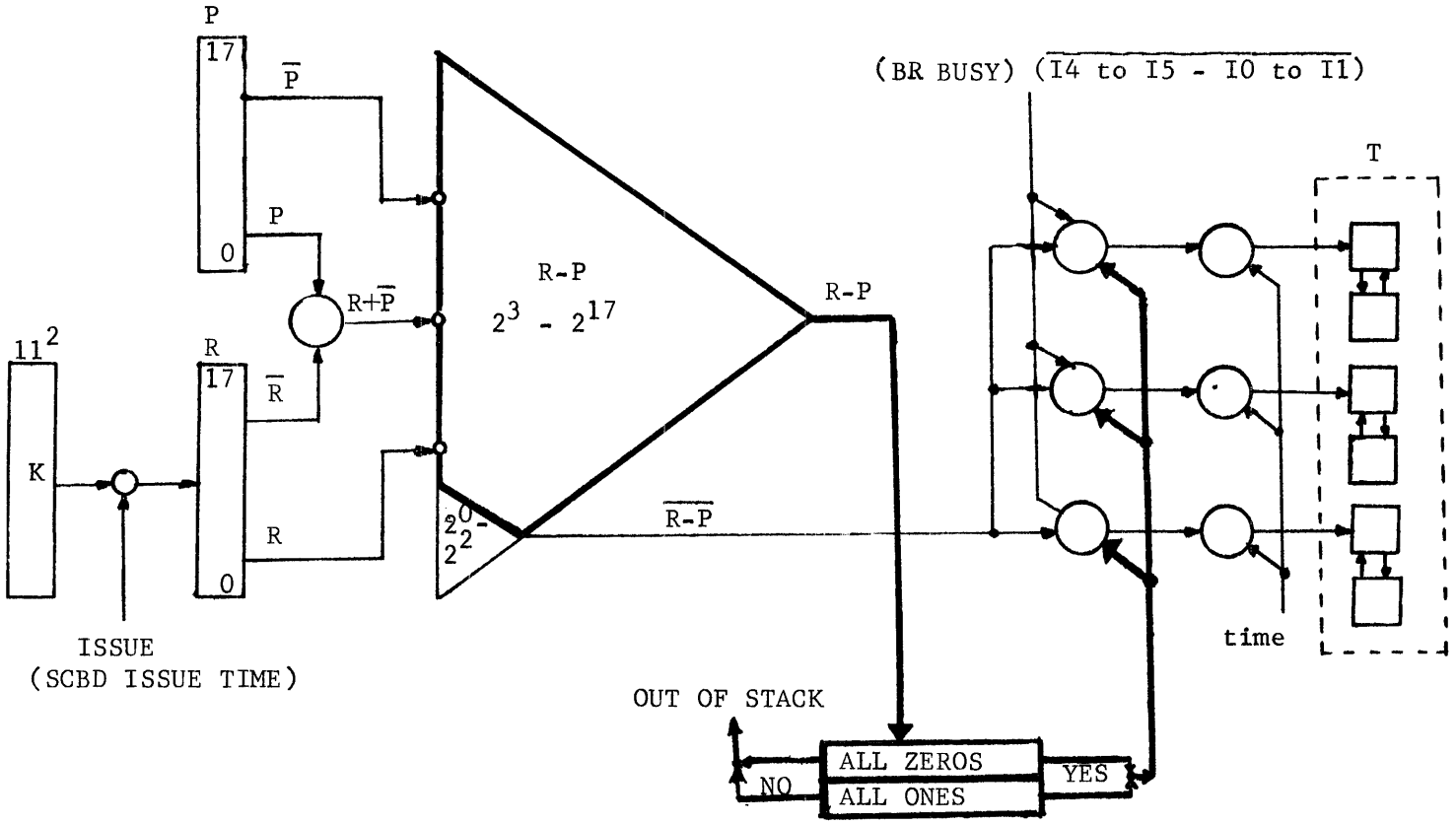

P
17
0
$\overline{P}$
P
R+$\overline{P}$
R
17
0
$\overline{R}$
R
11^2
K
ISSUE
(SCBD ISSUE TIME)
R-P
2^3 - 2^{17}
2^0 - 2^2
R-P
$\overline{R-P}$
(BR BUSY) ($\overline{I4 \text{ to } I5 - I0 \text{ to } I1}$)
T
time
OUT OF STACK
NO
ALL ZEROS
ALL ONES
YES

Figure 7.8-5

2) $\overline{L}$ - T TEST

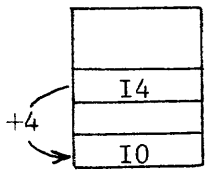
Even thougn the absolute value of R - P is less than or equal to seven, the "branch out of stack" condition may exist if there are not enough I registers before or after the I register containing the branch instruction. Assume, for example, the branch instruction is in I5 and branch of 6 places forward 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 $\overline{L}$ - 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 ($\overline{L}$ - 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 $\overline{L}$ - 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 $\overline{L}$ - T network.

In analyzing the result of the $\overline{L}$ - 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 $\overline{L}$ - T network subtracts the number of places we wish to branch from the I register number where the Branch instruction is located (i.e. $\overline{L}$ 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.



L = 3

L = 3

T = 4

$\overline{L}$ - T = 4 - 4 = 0

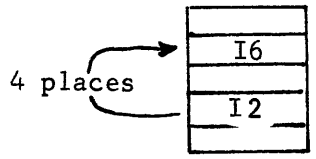
Thus, a difference greater than or equal to zero indicates the branch is forward less than the maximum number of places. (Incidentally, the condition $\overline{L}$ - T $\geq$ zero is sufficient condition to branch forward in the stack (loop) as we shall see shortly. A negative result ($\overline{L}$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

EAB $\Longrightarrow$ loop (in stack)

EAB $\Longrightarrow$ 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 $\overline{L}$.

Example: Branch backward 4 places from I2



L = 5
branch 4 places (T = 4 - true value)
$\overline{L}$ + T = 2 + 4 = 6

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

Note that a branch greater than 5 places backward from I2 would cause overflow (End Around Borrow). $\overline{L}$ + T = 2 + 6 = 10. Essentially the same thing is accomplished by subtracting the complement of the desired number of places to branch from $\overline{L}$. (Since adding can be accomplished by complementing and subtracting).

Using the same example as before:

L = 5 (I2)

T = 3 (Complement of # places to branch)

$\overline{L}$ - T = 2 - $\overline{3}$ = 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 $\overline{L}$ - 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 $\overline{L}$ - T network. This carry makes the $\overline{L}$ - T adder a two's complement network. It thus forms the quantity expressed as follows:

$\overline{L}$ - T - carry = 1

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

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

$\overline{L}$ - T = 1

The output of the $\overline{L}$ - 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 $\overline{L}$ - 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 I2 and a branch backward 5 places (to I7) is desired.

Then: L = 5, T = 2

<u>One's complement</u>:

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

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

<u>Two's complement</u>:

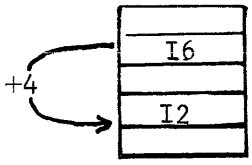
$\overline{L}$ - T - carry =

2 - 2 - 1 = 7

(The result is correct, indicating I7)

To summarize the $\overline{L}$ - T network, the following four cases are presented. The result of $\overline{L}$ - T will be used, only if the first condition ($|R - P| \leq 7$) is met.

1. <u>Branch Forward In Stack</u>



T = 4
all zeros to $\overline{\text{carry}}$

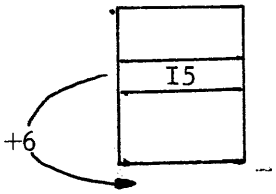
L = 1

$\overline{L}$ - T = 1

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

(all zeros) · ($\overline{EAB}$) $\Longrightarrow$ in stack

2. Branch Forward Out of Stack



T = 6
all zeros to $\overline{\text{carry}}$

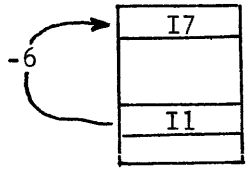
L = 2

$\overline{L}$ - T = 1

5 - 6 = 7 and EAB

(all zeros) • (EAB) $\Longrightarrow$ out of stack

3. Backward Branch In Stack



T = 1
all ones to carry

L = 6

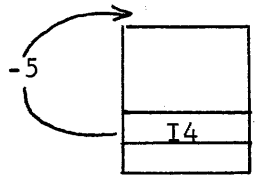
$\overline{L}$ - T = 1

1 - 1 - carry = 7 and EAB

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

Note: At this point the branch is $\overline{\text{out of stack}}$. Consideration #3 will determine whether or not the Branch is in the stack.

4. Backward Branch Out of Stack



T = 2
all ones to carry

L = 3

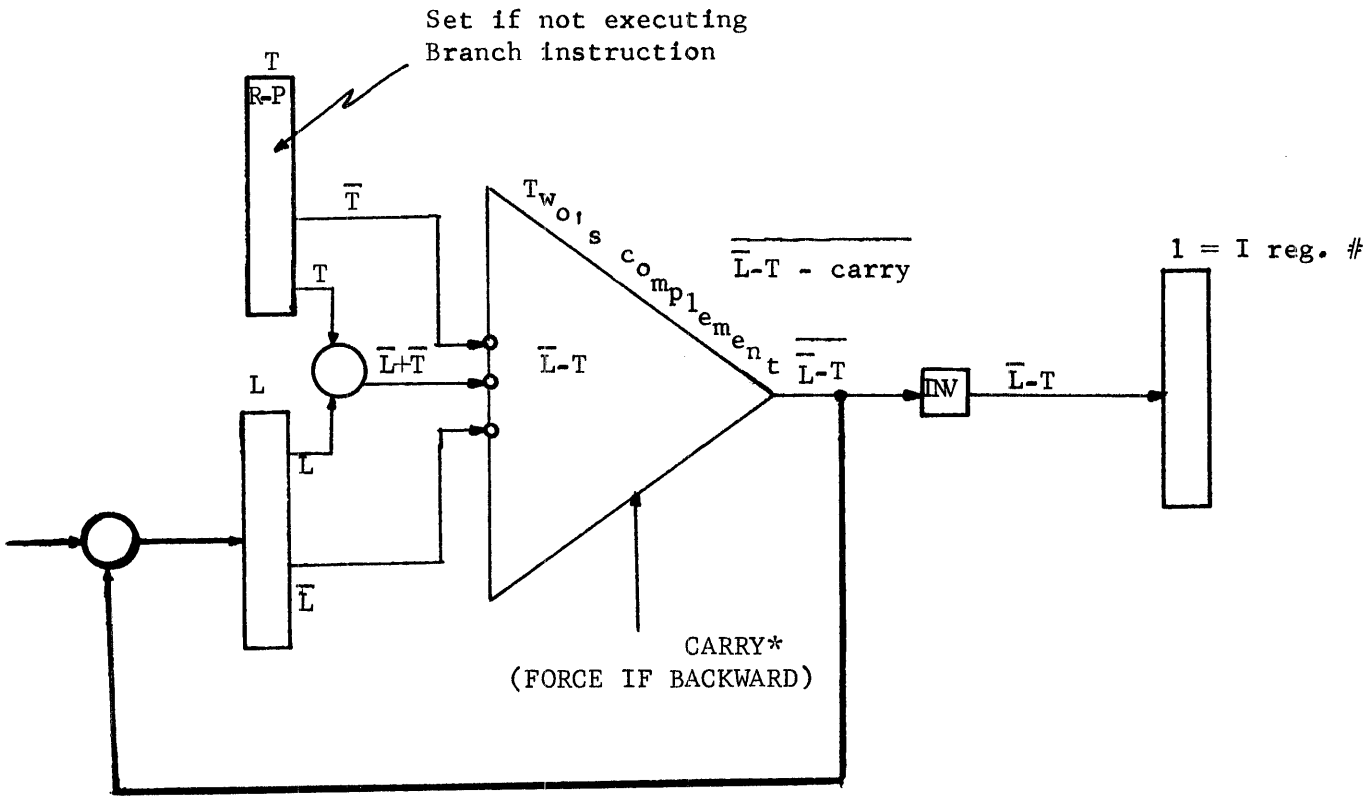
$\overline{L}$ - T = 1

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

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

Branch is out of stack.

BLOCK DIAGRAM -- $\overline{L}$ - T



*Makes result true if T was negative

Figure 7.8-6

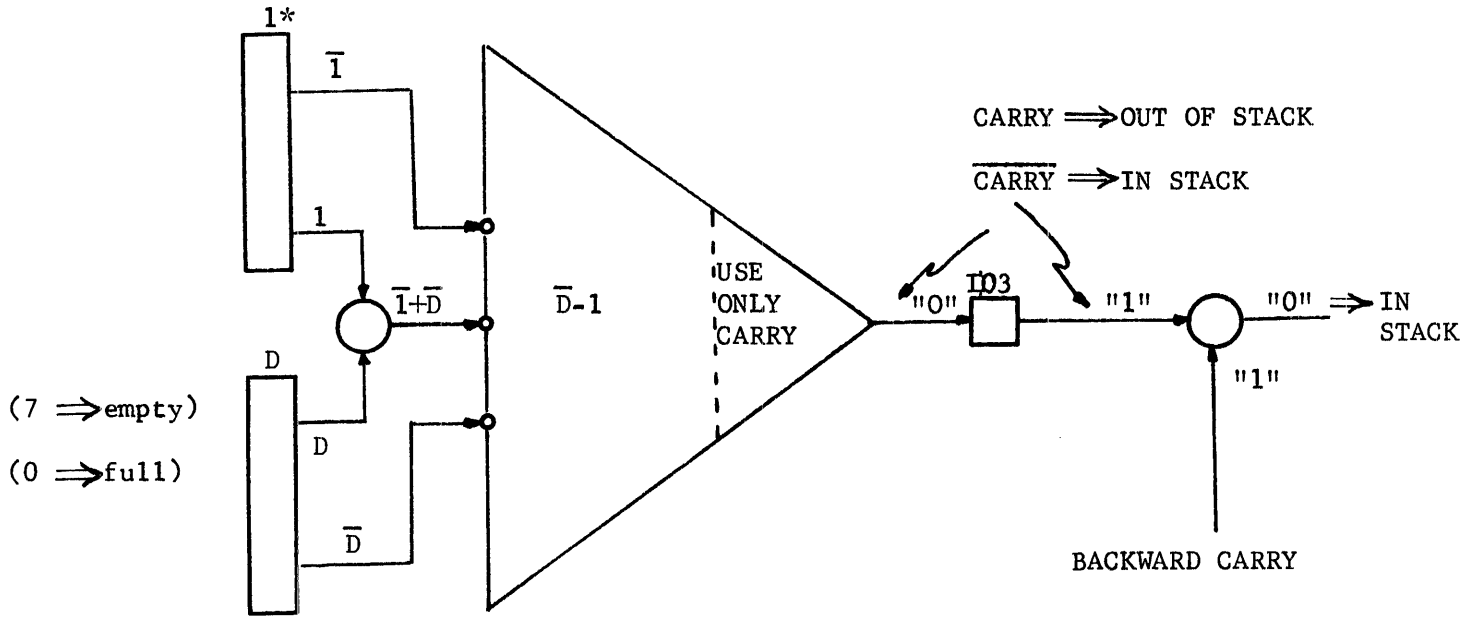
3) $\overline{D}$ - ($\overline{L}$ - T) TEST

From the previous examples it was seen that the $\overline{D}$ - ($\overline{L}$ - T) test is necessary for only case number 3 of the $\overline{L}$ - T test, although the test is performed whether or not the branch is backward. With this case it was determined that the branch is backward and within the limit of the physical I registers (1.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 I0 (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, $\overline{D}$ - ($\overline{L}$ -T), recall that the result of the $\overline{L}$ - 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, $\overline{D}$ 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 ($\overline{D}$) 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 ($\overline{L}$ - T) from the number of the register holding the first valid instruction ($\overline{D}$). If $\overline{L}$ - T is less than or equal to $\overline{D}$ 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 $\overline{L}$ - T exceeds $\overline{D}$, 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).

BLOCK DIAGRAM -- $\overline{D}$ - ($\overline{L}$-T)



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

Figure 7.8- 7

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

Figure 7.8-8 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.

IN STACK/OUT STACK TESTS

FLOW CHART

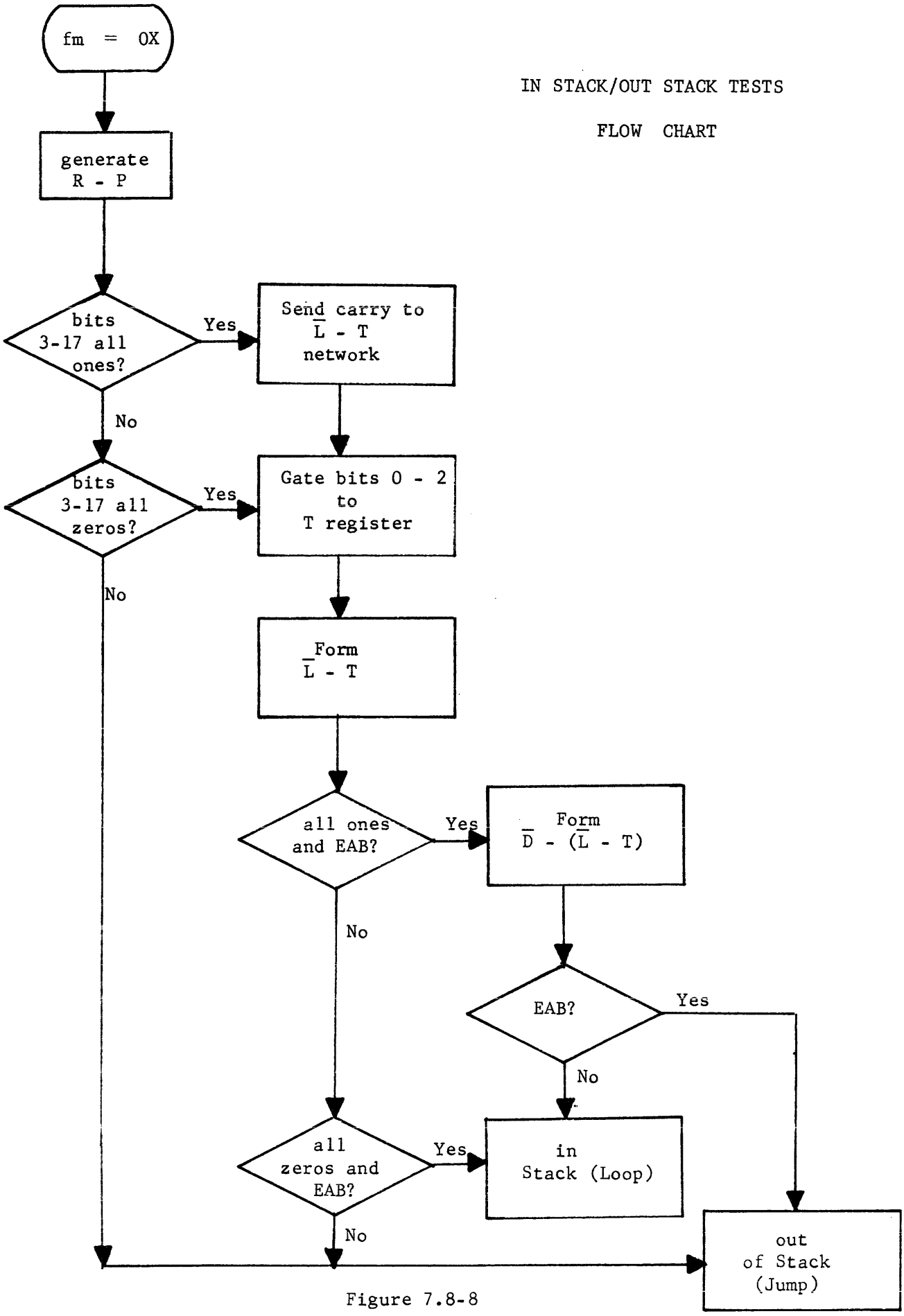


Figure 7.8-8

7.8.5 UNCONDITIONAL AND RETURN JUMPS

Unconditional Jump (fm = 02)

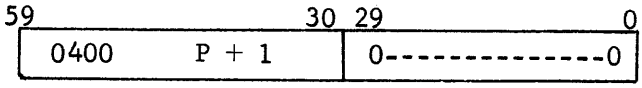
The unconditional jump uses very little of the Branch Units' logic. It, of course, does not use the "Condition Met" circuits nor does it use the "In Stack/Out Stack" tests (since the 02 is always out of the stack).

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 7.8-13 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 + Bi. 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:



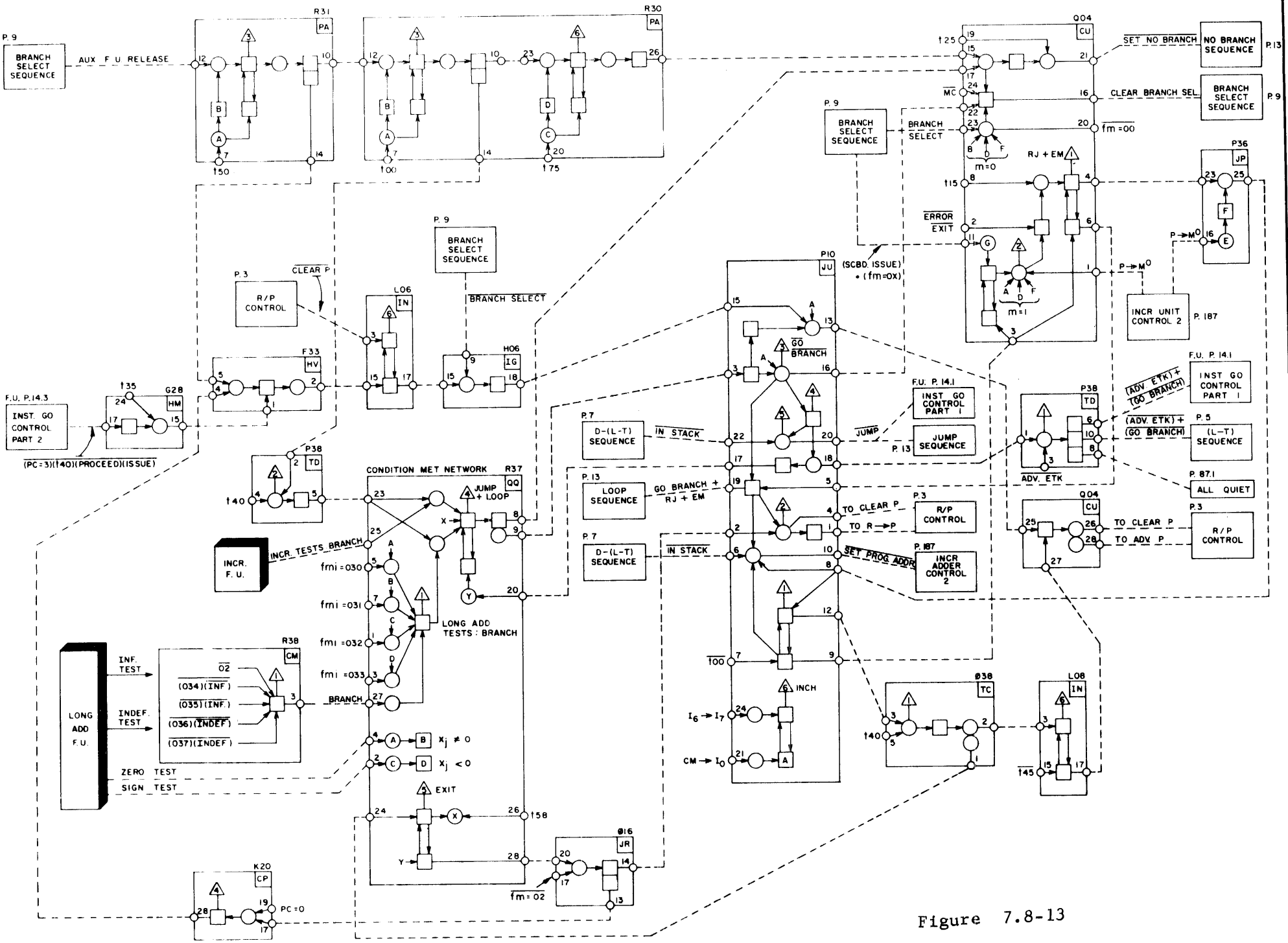

P. 9
BRANCH SELECT SEQUENCE
AUX F.U. RELEASE
R31
PA
R30
PA
Q04
CU
SET NO BRANCH
NO BRANCH SEQUENCE
P. 13
CLEAR BRANCH SEL
BRANCH SELECT SEQUENCE
P. 9
MC
BRANCH SELECT
fm=00
m=0
RJ + EM
P36
JP
ERROR EXIT
(SCBD. ISSUE) • (fm=0x)
P10
JU
P→M0
INCR UNIT CONTROL 2
P. 187
P. 3
R/P CONTROL
CLEAR P
L06
IN
BRANCH SELECT
H06
IG
F33
HV
F.U. P. 14.3
INST. GO CONTROL PART 2
G28
HM
(PC=3)(↑40)(PROCEED)(ISSUE)
P38
TD
CONDITION MET NETWORK
R37
QQ
JUMP + LOOP
INCR. F.U.
INCR. TESTS BRANCH
fmi=030
fmi=031
fmi=032
fmi=033
LONG ADD TESTS: BRANCH
BRANCH
LONG ADD F.U.
INF. TEST
INDEF. TEST
ZERO TEST
SIGN TEST
R38
CM
EXIT
K20
CP
PC=0
fm=02
Ø16
JR
P. 7
D-(L-T) SEQUENCE
IN STACK
JUMP
P. 13
JUMP SEQUENCE
F.U. P. 14.1
INST GO CONTROL PART 1
GO BRANCH
P. 13
LOOP SEQUENCE
GO BRANCH + RJ + EM
TO CLEAR P
TO R→P
SET PROG ADDR
INCR ADDER CONTROL 2
P. 187
ADV. ETK
(ADV. ETK) + (GO BRANCH)
P. 5
(L-T) SEQUENCE
P. 87.1
ALL QUIET
TO ADV P
INCH
Ø38
TC
L08
IN
I6 → I7
CM → I0

Figure 7.8-13

It then transfers program control to location K + 1.

To perform these operations, the following events take place (Refer to Figure 7.8-10).

1. Issuance of instructions is stopped as with any 0X instruction.
2. The output of the incrementer (P + L) is sent to the S register with the following gate:

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

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

   (fm = 01)(P to M^0)(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) =

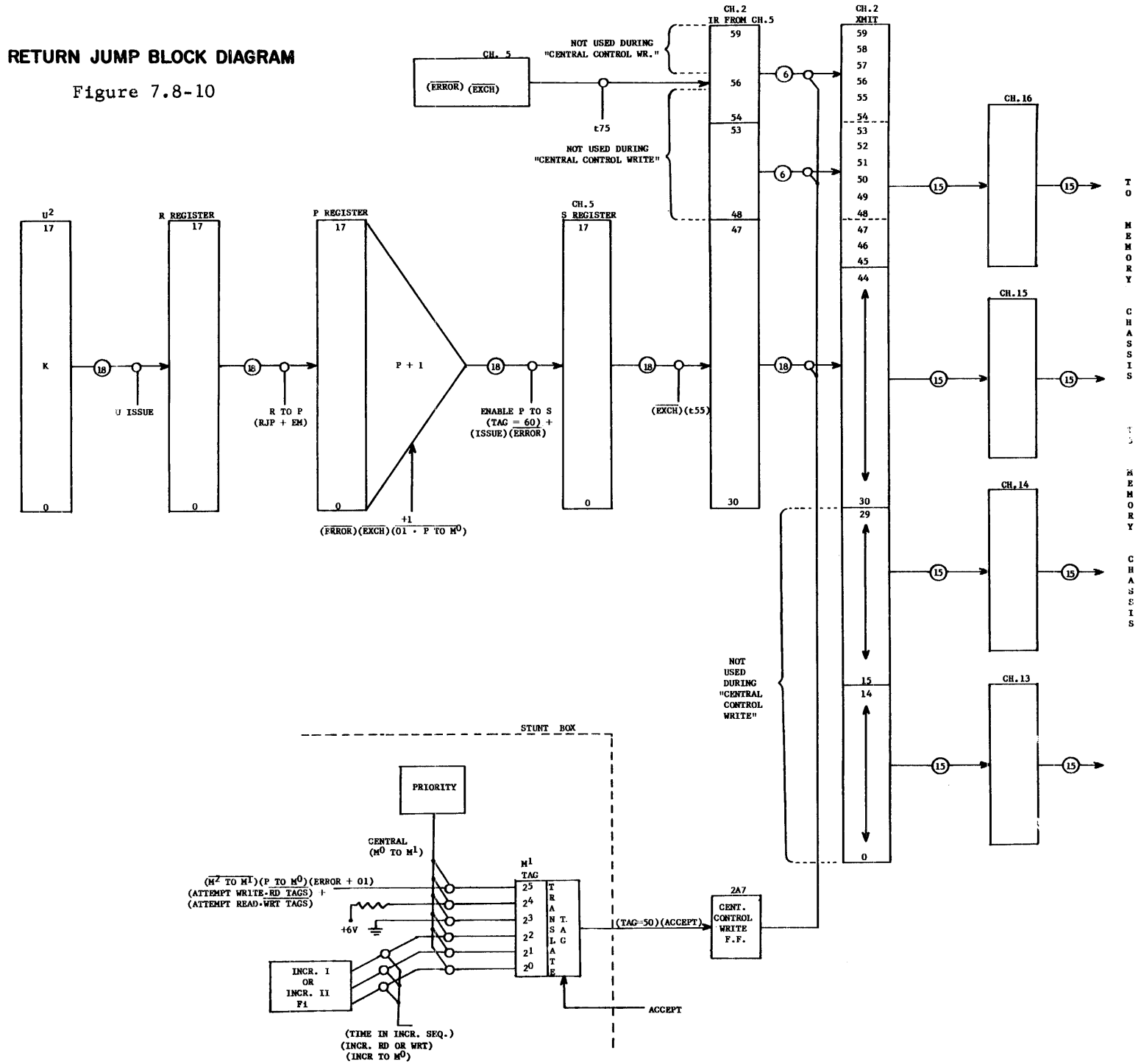
59 — 48	47 — 30	29 — 0
0400	P + 1	

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, TP1), 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).

RETURN JUMP BLOCK DIAGRAM

Figure 7.8-10



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 7.8-13). If the condition is met, the "Go Branch" (Jump or Loop) flip/flop is set (R37, TP4). If the condtion 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 cleares 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 $\overline{L}$ - 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.

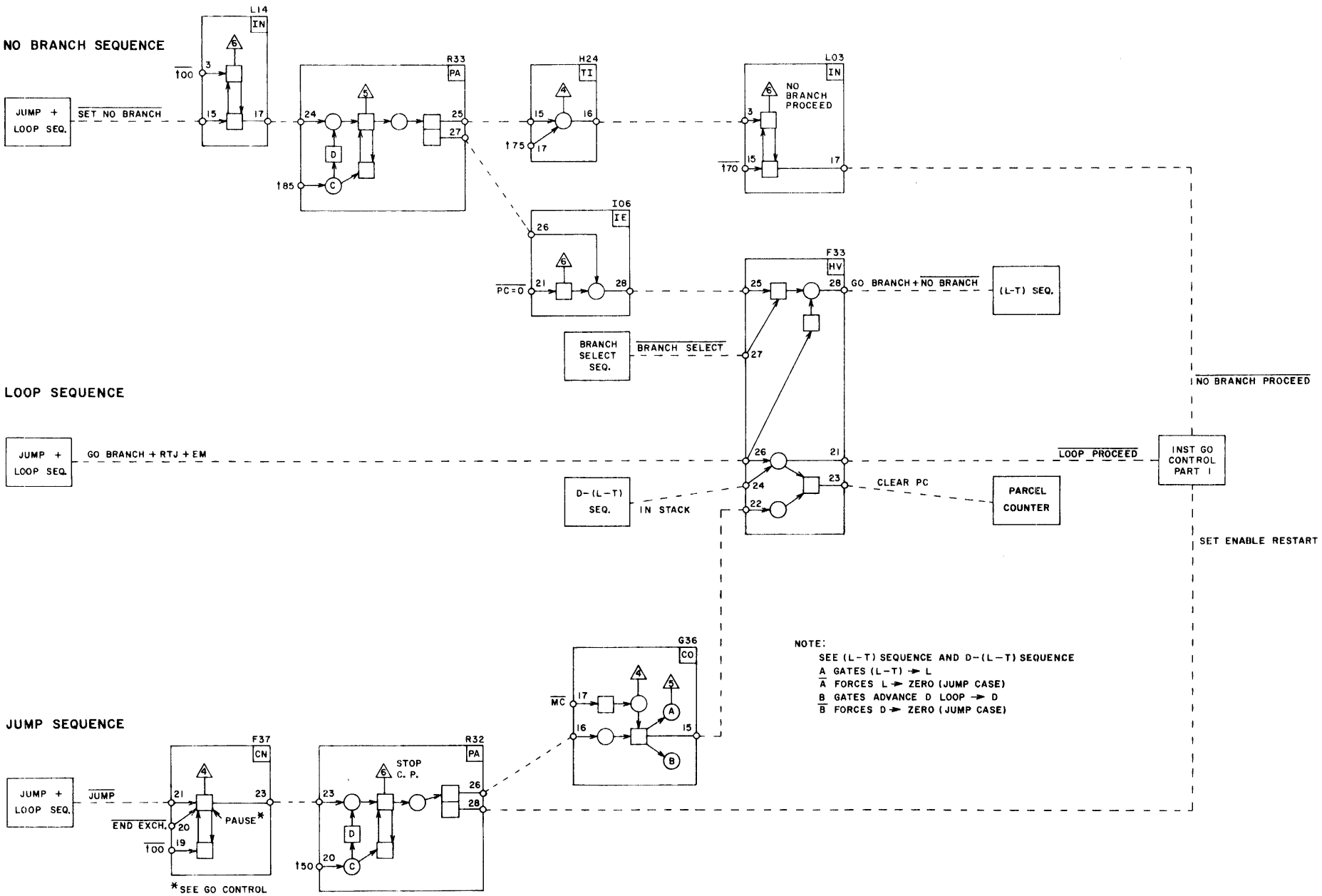
7.8.7 LOOP SEQUENCE

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 7.8-13 , 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 "zero" 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 (C.E. Diagrams, sheet 98) via F33, pin 28 (Figure 7.8-12). Term "G" gates the output of the $\overline{L}$ - T network into the L register for either the Jump or Loop case.

NOTE: As will be seen, the Jump sequence will set L = 7 after setting a new value of L. Thus, no problem arises by setting the new L for both the Loop and Jump cases. (See Section 7.8.8)



NO BRANCH /LOOP/ JUMP SEQUENCE
Figure 7.8-12

P10, pin 2 (Figure 7.8-13) says $(\overline{RTJ})(\overline{EM})(\overline{02})$. 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 7.8-12).

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.

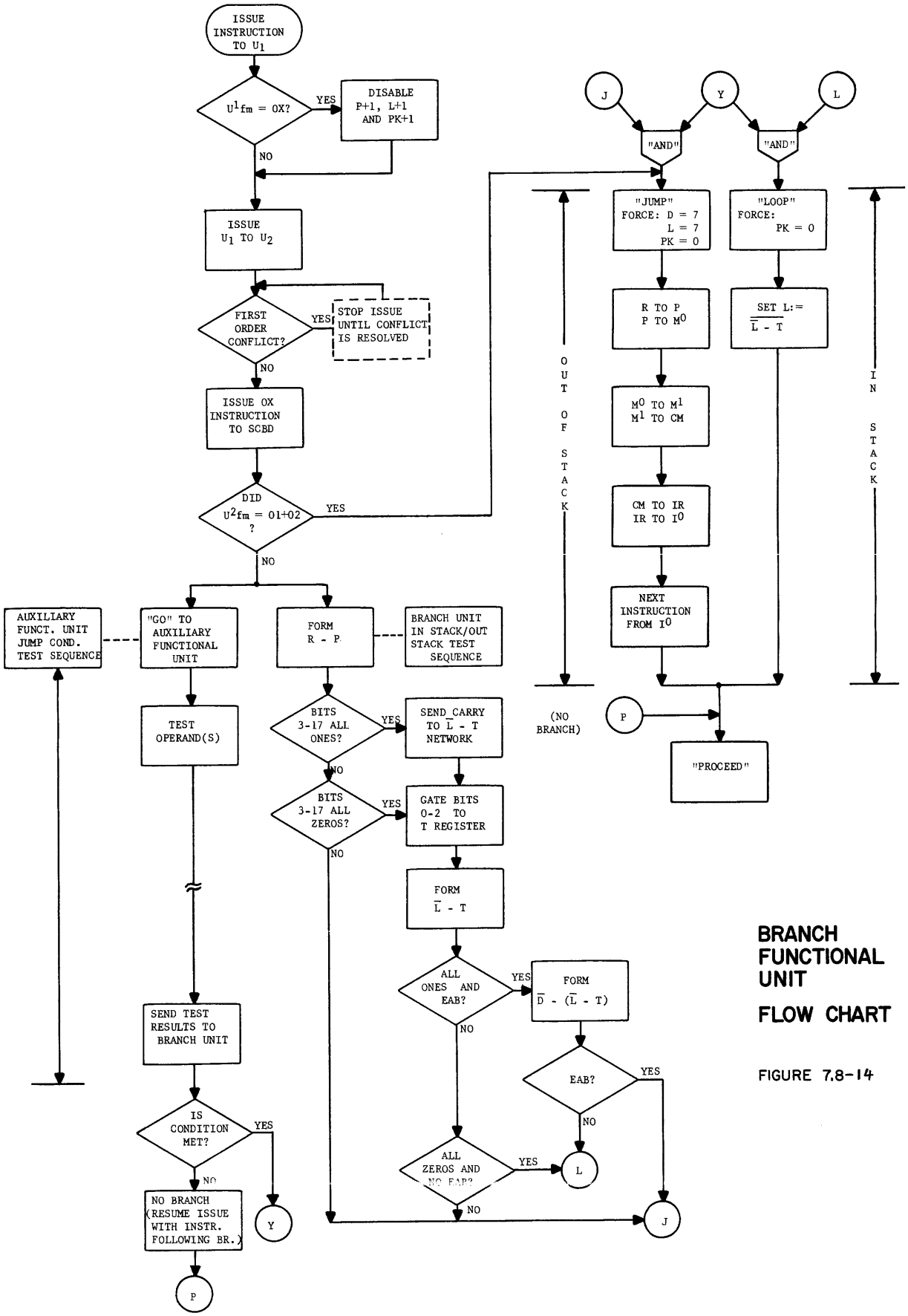
7.8.8 JUMP SEQUENCE

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 $\overline{RTJ + EM}$ 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 7.8-13) 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.



BRANCH FUNCTIONAL UNIT

FLOW CHART

FIGURE 7.8-14

APPENDIX A

6000 SERIES FLOATING POINT

APPENDIX A

6000 SERIES FLOATING POINT

Floating point hardware provides the ability to express a number in the general form:

$$K \cdot B^n$$

where:

K = coefficient

B = base number

n = power to which the base number is raised

This concept is opposed to fixed point hardware, where a number is expressed without exponent. Fixed point hardware places the burden of exponent expression and manipulation upon the software.

To express numbers of the above general form, provision is made to represent four properties:

1) Magnitude of the coefficient
2) Sign of the coefficient
3) Magnitude of the exponent (power)
4) Sign of the exponent

The base number, B, is assumed to be 2. Hence there is no need to express the base number. In other words, the computer assumes the exponent to be a power of 2.

The 6000 Series 60-bit floating point format is shown in figure A - 3.

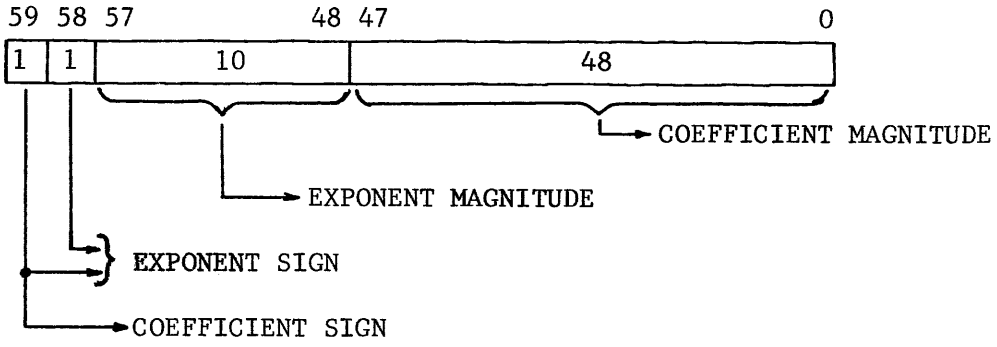


figure A - 3

The lower 48-bits are reserved for expressing the magnitude of the coefficient. The computer assumes the binary point to be to the right of the coefficient, thereby providing a 48-bit integer coefficient which is equivalent to about 15 decimal digits. Bit 2^{59} is used to express the sign of the coefficient. The following rules apply to signed coefficients:

SIGN	2^{59} =	MAGNITUDE (bits 0 - 47)
POSITIVE	0	True (uncomplemented) form
NEGATIVE	1	Complement form

Bits 48 - 57 are used to express the magnitude of the exponent. Hence, the absolute value of the exponent may be as large as $1777_{(8)}$. The sign of the exponent is expressed by the combination of bits 2^{58} and 2^{59}, according to the following rules:

If $2^{58} \neq 2^{59}$, the exponent sign is POSITIVE.

if $2^{58} = 2^{59}$, the exponent sign is NEGATIVE.

Figure A - 2 summarizes the configurations of bits 2^{58} and 2^{59}, and the implications, regarding signs, of the possible combinations.

2^{59}	2^{58}	COEFFICIENT SIGN	EXPONENT SIGN
0	1	POSITIVE	POSITIVE
0	0	POSITIVE	NEGATIVE
1	0	NEGATIVE	POSITIVE
1	1	NEGATIVE	NEGATIVE

figure A-2

PACKING

Packing refers to the conversion of numbers in the form, $K \cdot B^n$, into floating point format. The following rules apply to this process.

A. POSITIVE COEFFICIENT

1. Place the coefficient magnitude in positions 47 - 0.
2. Make bit 59 = 0.
3. IF EXPONENT IS POSITIVE,
   a. Make bit 58 = 1.
   b. Place the exponent magnitude in positions 57 - 48.
4. IF EXPONENT IS NEGATIVE,
   a. Make bit 58 = 0.
   b. Complement the exponent magnitude and place in positions 57 - 48.

B. NEGATIVE COEFFICIENT

1. Pack according to the four rules for positive coefficients.
2. THEN, complement all 60-bits.

EXAMPLE # 1 - POSITIVE COEFFICIENT AND POSITIVE EXPONENT:

PROBLEM: Pack the number: $+3427 \cdot 2^{+26}$

SOLUTION:

1) Rule A1: Coef. Mag. = ____0000000000003427

2) Rule A2: Coef. Sign = 0___0000000000003427

3) Rule A3a: Exp. Sign = 2___0000000000003427

4) Rule A3b: Exp. Mag. = 20260000000000003427

The result of step # 4 yields the packed quantity.

EXAMPLE # 2 - POSITIVE COEFFICIENT AND NEGATIVE EXPONENT:

PROBLEM: Pack the number: $+3427 \cdot 2^{-26}$

SOLUTION: ,

1) Rule A1: Coef. Mag. = ____0000000000003427

2) Rule A2: Coef. Sign = 0___0000000000003427

3) Rule A4a: Exp. Sign = 0___0000000000003427

4) Rule A4b: Exp. Mag. = 17510000000000003427

The result of step # 4 yields the packed quantity.

EXAMPLE # 3 - NEGATIVE COEFFICIENT AND POSITIVE EXPONENT:

PROBLEM: Pack the number: $-3427 \cdot 2^{+26}$

SOLUTION:

1) Rule B1: Use rules A1 through A4 first.

2) Rule A1: Coef. Mag. = ____0000000000003427

3) Rule A2: Coef. Sign = 0___0000000000003427

4) Rule A3a: Exp. Sign = 2___0000000000003427

5) Rule A3b: Exp. Mag. = 20260000000000003427

6) Rule B2: Complement = 57517777777777774350

The result of step # 6 yields the packed quantity.

EXAMPLE # 4 - NEGATIVE COEFFICIENT AND NEGATIVE EXPONENT:

PROBLEM: Pack the number: $-3426 \cdot 2^{-26}$

SOLUTION:

1) Rule B1: Use rules A1 through A4 first.

2) Rule A1: Coef. Mag. = ____0000000000003427

3) Rule A2: Coef. Sign = 0___0000000000003427

4) Rule A4a: Exp. Sign = 0___0000000000003427

5) Rule A4b: Exp. Mag. = 1751000000000003427

6) Rule B2: Complement = 6026777777777774350

The result of step # 6 yields the packed quantity.

In examples 3 and 4, check the sign of the coefficient and exponent as given by the packed quantity. In example 3, the upper octal is 5, or $101_{(2)}$. According to figure A.3-2 this implies a negative coefficient and a positive exponent, which are the correct signs of the original number. In example 4, the upper octal is 6, or $110_{(2)}$. According to figure A.3-2 this implies a negative coefficient and a negative exponent, which are the correct signs of the original number.

Although the above process is not a difficult one, it is somewhat cumbersome. A short-cut method of packing exponents can be derived by considering the representation of negative and positive zero exponents. Assuming a positive coefficient, zero exponents are packed as follows:

POSITIVE ZERO EXPONENT = 2000X----------X

NEGATIVE ZERO EXPONENT = 1777X----------X

Since positive exponents are expressed in true form, start with a "bias" of 2000 (positive zero) and add the magnitude of the exponent. The range of positive exponents is:

0000 through 1777

Or, in packed form:

2000 through 3777.

Negative exponents are expressed in complement form. Hence, start with a bias of 1777 (negative zero) and subtract the magnitude of the exponent. The range of negative exponents is:

-0000 through -1777

Or, in packed form:

1777 through 0000.

The rules for packing may now be simplified as follows:

A. POSITIVE COEFFICIENT

1. Place the coefficient magnitude in positions 47 - 0.
2. If the exponent is positive, add the exponent magnitude to 2000.
3. If the exponent is negative, subtract the exponent magnitude from 1777.

B. NEGATIVE COEFFICIENT

1. Pack according to the rules for positive coefficients.
2. THEN, complement all 60-bits.

UNPACKING

As the name implies, unpacking is the process opposite to packing. It refers to the conversion of floating point formats into numbers of the

form, $K \cdot B^n$. The following rules apply to this process:

A. Check the sign of the coefficient (bit 59).

1. If positive, take the lower 48-bits as the coefficient magnitude.

2. If negative,

a. Complement all 60-bits.

b. Take complemented lower 48-bits as the coefficient magnitude.

B. Check the complemented upper 12-bits.

1. If in the range, 2000 - 3777, subtract 2000 (bias) to obtain the magnitude of the POSITIVE exponent.

2. If in the range, 0000 - 1777, subtract <u>from</u> 1777 (bias) to obtain the magnitude of the NEGATIVE exponent.

These rules may be proven by working examples 1 - 4 in reverse to obtain the original mathematical expression.

<u>RANGE DEFINITIONS</u>

The preceding discussion presented the format and conversion methods of 6000 Series floating point. The 6000 Series computers provide the capability of representing a 48-bit integer coefficient multiplied by 2 raised to a power in the range $\pm 1777_{(8)}$.

In Octal: $7777777777777777. \cdot 2^{\pm 1777}$

The decimal equivalent is:

$281474976710655. \cdot 10^{\pm 307.95369}$

OVERFLOW

A result with an exponent so large that it reaches or exceeds the upper limit of 3777 (positive) or 4000 (negative) is treated as an infinite quantity. In the case of positive infinity, a coefficient of all zeros and an exponent of octal 3777 are packed. For negative infinity, a coefficient of all zeros and an exponent of octal 4000 are packed. An optional exit is provided when an infinite operand is detected since its use may generate further error conditions. (Refer to Appendix B.)

UNDERFLOW

A result, the exponent of which is less than the lower limit of octal 0000 (positive) or 7777 (negative) is treated as a zero quantity. This condition will cause a zero exponent and coefficient to be packed. No exit is provided for underflow. A result with an exponent of octal 0000 and a coefficient that is not zero is a non-zero quantity and is packed with an octal 0000 exponent and the non-zero coefficient.

INDEFINITE OPERANDS

Use of infinity, zero, or indefinite operands may produce an indefinite result. An exponent of octal 1777 and a zero coefficient are packed in this case. An optional exit is provided.

Hence, octal exponent configurations of 1777, 3777, 4000, and 6000 have special meaning and should not be used in representing numbers in floating point.

In summary, the special operand forms in octal are:

3777X——X (positive infinity)

4000X——X (negative infinity)

1777X——X (positive indefinite)

6000X——X (negative indefinite)

00000——0 (positive zero)

77777——7 (negative zero)

Whenever an operand in one of these six special forms is used as a source operand, only the following octal words can occur as results:

37770——0 (positive infinity)

40000——0 (negative infinity)

17770——0 (positive indefinite)

00000——0 (positive zero)

Note that in these cases, the 48 least significant bits of the result are zeros. For a detailed list of infinite, zero, and indefinite forms, refer to Appendix B.

SINGLE/DOUBLE PRECISION

The 6000 Series floating point add, subtract and multiply instructions are capable of producing single or double precision results. Single precision results may be rounded or unrounded; double precision opcodes always return unrounded results. Floating point divide instructions generate only rounded or unrounded single precision results.

FLOATING ADD OPERATIONS

Before addition of floating point numbers takes place, the exponents are equalized by shifting the coefficient of the smaller exponent to the right a number of places given by the difference of the two exponents. The shifted and unshifted coefficients* are added and a 96-bit sum is generated as illustrated in the following example:

```
GIVEN:    FX6 = X1 + X2
          (X1) =  2072 4273000012340772
          (X2) =  2057 5230000023457610
```

The Add logic forms:

```
X1 (unshifted)     =  4273000012340772.
X2 (RS 15 places)  =       52300000234.57610
96-bit Result      =  4273052312341226.5761000000000000
```

In effect, right shifting the coefficient of the smaller exponent (which decreases the coefficient magnitude) increases that exponent to equal the larger. Hence, the exponent of the 96-bit sum is the larger of the two original exponents and the binary point in the example is correctly positioned at the center of the result. Since unrounded single precision is specified (FX6), the upper 48-bits of the sum and the larger exponent will be taken as the final result:

```
(X6) =  2072 4273052312341226
```

Had double precision been specified (DX6), the final result would be the lower 48-bits of the sum and the larger exponent minus 60_8 (48_{10}):

```
(X6) =  2012 5761000000000000
```

---

* Neither coefficient will be shifted if the original exponents are equal.

Packing the lower 48-bits of the sum effectively moves the binary point 48 places to the right, which increases the coefficient magnitude. To compensate, the exponent is decreased by 48 (60_8) during double precision.

If rounded single precision were specified (RX6), a "one" would be placed below bit 2^0 of the unshifted operand before adding. This, in effect, adds $\frac{1}{2}$ and will cause a carry into bit 2^0 if the fractional portion of the shifted operand is greater than or equal to $\frac{1}{2}$. To illustrate, the original example with round specified follows:

The Add logic forms:

```
X1 (unshifted)     =  4273000012340772.4
X2 (RS 15 places)  =       52300000234.57610
96-bit result      =  4273052312341227.1761000000000000
```

The final result in this case is:

(X6) = 2072 4273052312341227

FLOATING MULTIPLY OPERATIONS

Multiplication of floating point numbers requires the addition of the exponents and the multiplication of two 48-bit coefficients. A coefficient product of 96-bits, with the binary point below bit 2^0, is formed. If both original operands are normalized, the result will also be normalized.

GIVEN: FX7 = X3 * X4
(X3) = 2023 4253000000000002
(X4) = 2027 5300000000000014

The Multiply logic forms:

```
X3                                  4253000000000002.
X4                                  5300000000000014.
96-bit result  =  27227100000000074604000000000030.
```

Since this result is not normalized, it is left shifted one place:

LS 1 place = 5645620000000017141000000000060

Since single precision was specified (FX7), the upper 48-bits of the result will be packed as the final coefficient. This effectively moves the binary point to the left 48 places, decreasing the magnitude of the coefficient. Hence, 60_8 is added to the sum of the exponents. To compensate for the left shift required to normalize, one is subtracted from the exponent. The single precision result is:

(X7) = 2131 5645620000000017

(NOTE: Final exponent = 2023 + 2027 + 60 - 1 = 2131)

Had double precision been specified (DX7), the lower 48-bits of the product and the sum of the exponents minus one (compensating for the left shift to normalize) would be taken as the result:

(X7) = 2051 1410000000000060

If rounded single precision was specified (RX7), a "one" would be added to bit 2^{46} of Xk (X4 in this example) on the first iteration of the multiply step. A rounded, 96-bit coefficient product would be generated and the upper 48 bits, along with the sum of the exponents plus 57_8 would be taken as the result:

(X7) = 2131 5645620000000020

In the above example a $\frac{1}{2}$ round occurred because the 96-bit coefficient was left shifted one place to normalize. Had the product already been normalized, the left shift would not occur. The result would be

rounded by ¼ (since the round bit was entered in position 2^{46}) and the final exponents would all be greater by one.

FLOATING DIVIDE OPERATIONS

Division of floating point numbers requires the subtraction of the exponents and the division of two 48-bit coefficients. Double precision division is not provided and a remainder cannot be retrieved. Since the divide hardware produces a quotient in the range, 1.7777777777777777 through 0.0000000000000000, the ratio of Xj to Xk must always be less than 2 to 1. If this requisite is not met, the resulting quotient will be meaningless.*

If the quotient is of the form, 0.X----------X (the ratio of Xj to Xk is less than 1 to 1), the lower 48-bits of a 49 bit quotient are taken as the coefficient of the result:

GIVEN: FX6 = X4 / X5
(X4) = 2057 4400000000000015
(X5) = 2032 6000000000000000

The Divide logic forms:

$$\frac{4400000000000015}{6000000000000000} = 0.6000000000000002$$

---

*It is suggested that the Xj and Xk operands always be normalized before executing a Divide opcode. This will eliminate the possibility of the Dividend/Divisor ratio (Xj/Xk) being greater than or equal to 2 to 1.

Packing into floating point format effectively moves the binary point 48 places to the right, increasing the coefficient magnitude. 60_8 is therefore subtracted from the difference of the exponents and the final result is:

(X6) = 1744 6000000000000002

(NOTE: Final exponent = 2057 - 2032 - 60 = -33 = 1744)

If the quotient is of the form, 1.X----------X (the ratio of Xj to Xk is 1 to 1 or greater, but less than 2 to 1), the upper 48-bits of a 49-bit quotient are taken as the coefficient of the result:

GIVEN: FX7 = X1 / X2
(X1) = 2016 7000000000000000
(X2) = 2025 4000000000000000

The Divide logic forms:

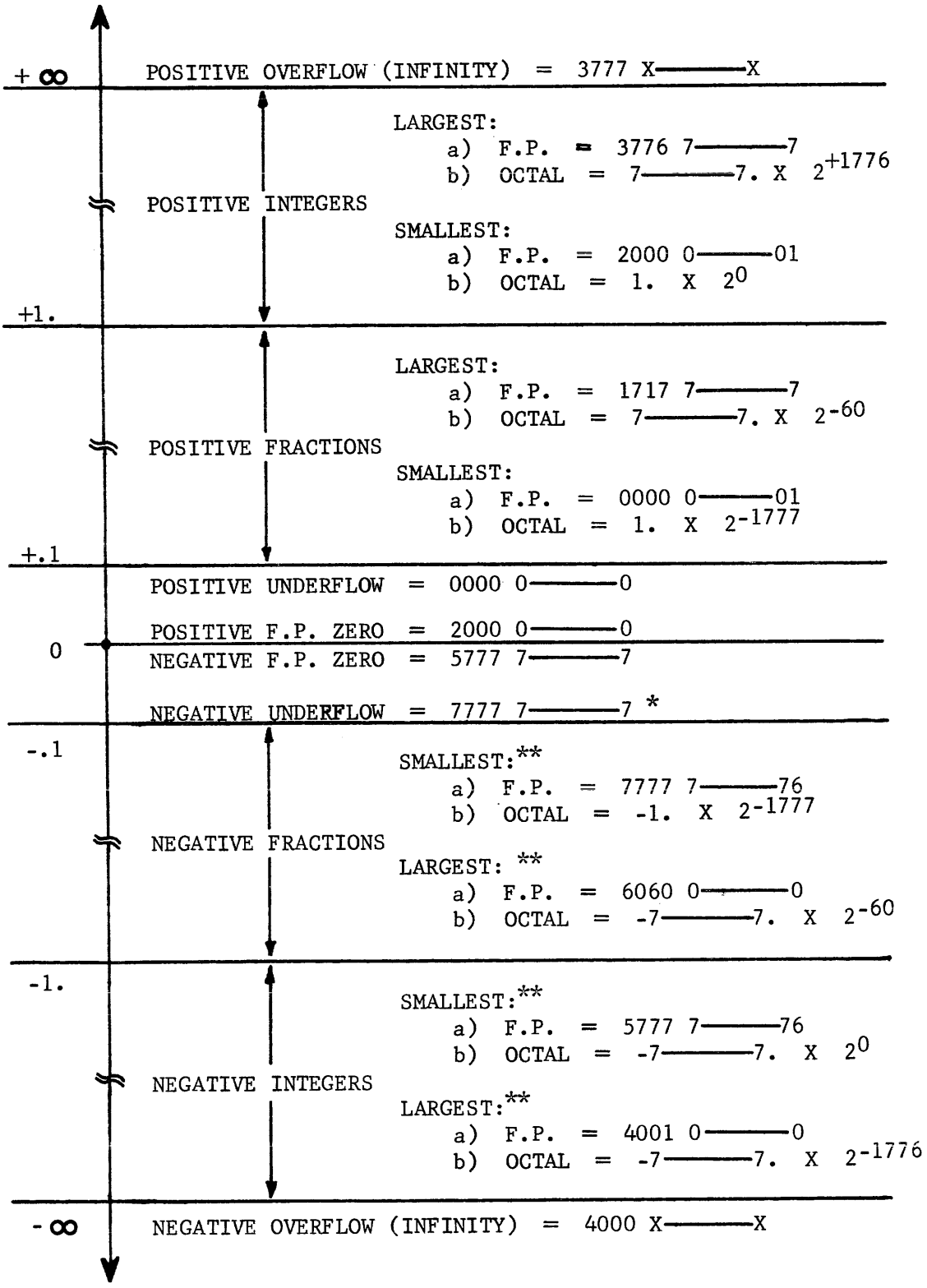
$$\frac{7000000000000000}{4000000000000000} = 1.6000000000000000$$

Packing into floating point format effectively moves the binary point 47 places to the right. 57_8 is therefore subtracted from the difference of the exponents and the final result is:

(X7) = 1711 7000000000000000

(NOTE: Final exponent = 2016 - 2025 - 57 = -66 = 1711)

6000 SERIES FLOATING POINT REPRESENTATION



* The machine packs all zeros (positive underflow) for this case.

** In absolute value.

APPENDIX B

NONSTANDARD OPERAND FORMS

NON-STANDARD FLOATING POINT ARITHMETIC

The following is a tabulation of operations (Add, Subtract, Multiply, Divide) using various combinations of operands.

KEY:

OPERANDS			RESULTS		
+0	=	0000 X...X	0	=	0000 0...0
-0	=	7777 X...X	IND	=	1777 0...0
+ ∞	=	3777 X...X	+ ∞	=	3777 0...0
- ∞	=	4000 X...X	- ∞	=	4000 0...0
+IND	=	1777 X...X			
-IND	=	6000 X...X			
W	=	Any word except ± ∞ , ±IND			
N	=	Any word except ± ∞ , ±IND, or ±0			

ADD

Xi=Xj+Xk

(Instructions 30, 32, 34)

		Xk			
		W	+ ∞	- ∞	±IND
Xj	W	-	+ ∞	- ∞	IND
	+ ∞		+ ∞	IND	IND
	- ∞		∞	- ∞	IND
	±IND				IND

SUBTRACT

Xi=Xj-Xk

(Instructions 31, 33, 35)

		Xk			
		W	+ ∞	- ∞	±IND
Xj	W	-	- ∞	+ ∞	IND
	+ ∞	+ ∞	IND	+ ∞	IND
	- ∞	- ∞	- ∞	IND	IND
	±IND	IND	IND	IND	IND

MULTIPLY

Xi=Xj*Xk

(Instructions 40, 41, 42)

		Xk						
		+N	-N	+0	-0	+∞	-∞	±IND
Xj	+N	-	-	0	0	+∞	-∞	IND
	-N		-	0	0	-∞	+∞	IND
	+0			0	0	IND	IND	IND
	-0				0	IND	IND	IND
	+∞					+∞	-∞	IND
	-∞						+∞	IND
	±IND							IND

DIVIDE

Xi=Xj/Xk

(Instructions 44, 45)

		Xk						
		+N	-N	+0	-0	+∞	-∞	±IND
Xj	+N	-	-	+∞	-∞	0	0	IND
	-N	-	-	-∞	+∞	0	0	IND
	+0	0	0	IND	IND	0	0	IND
	-0	0	0	IND	IND	0	0	IND
	+∞	+∞	-∞	+∞	-∞	IND	IND	IND
	-∞	-∞	+∞	-∞	+∞	IND	IND	IND
	±IND	IND	IND	IND	IND	IND	IND	IND

COMMENT SHEET

6600 CENTRAL PROCESSOR, Volume II

Publication Number 60239700

FROM: Name: _____

Business
Address: _____

COMMENTS: (Describe errors, suggested additions or deletions, and include page numbers, etc.)

CONTROL DATA INSTITUTES

3255 Hennepin Avenue So.
MINNEAPOLIS, MINNESOTA
55408

5630 Arbor Vitae Street
LOS ANGELES, CALIFORNIA
90045

3717 Columbia Pike
ARLINGTON, VIRGINIA
22204

CONTROL DATA
COMPUTER TRAINING SCHOOL
66 West 12th Street
NEW YORK, NEW YORK
10011

60 Hickory Drive
Bear Hill Industrial Park
WALTHAM, MASSACHUSETTS
02154

Exchange Park Garden Mall
DALLAS, TEXAS
75235

23775 Northwestern Highway
SOUTHFIELD, MICHIGAN
48075

Bockenheimer Landstr. 10
6000 FRANKFURT/M.
GERMAN FEDERAL REPUBLIC

60239700

CONTROL DATA
CORPORATION