

KASTNER'S

MEMOS

4X TI-ASC

MEMORANDUM

4 April 1972

TO Hardware Design (15)
System Engineering
Wayne Winkelman (4)

FROM Bill Kastner

SUBJECT NEW INSTRUCTIONS FOR THE
TIMES-FOUR CENTRAL PROCESSOR

A description of all new times-four CP instructions is attached. These instructions are listed following:

1. FORK, Fork.
2. JOIN, Join.
3. PB, Prepare to Branch.
4. LEM, Load Arithmetic Exception Mask and Condition.
5. LRL, Load Arithmetic Register Right Half from Alpha Left Half.
6. STRL, Store Arithmetic Register Right Half to Alpha Left Half.
7. SCLK, Store 32-bit Fixed Point Clock.
8. Select on Equal, Select on Not Equal (Vector).
9. Replace on Equal, Replace on Not Equal (Vector).

Bill Kastner

BILL KASTNER

WDK:jc

Attachment

ATTACHMENT

<u>Fork</u>		FORK	Mnemonic code
1 → Fork Indicator		9A	Op code

The FORK instruction is an advisory type instruction to the IPU control. Execution of the FORK instruction sets the fork indicator bit within the IPU control and allows subsequent vector or scalar instructions to proceed to execution independently. In the times-four CP, this means that any combination of vector or scalar instructions can be in execution simultaneously in each of the four MBU-AU pairs. Refer to the write-up describing FORK and JOIN control for further details on the effect of this instruction.

<u>Join</u>		JOIN	Mnemonic code
0 → Fork Indicator		9B	Op code

The JOIN instruction is an advisory type instruction to the IPU control. Execution of the JOIN instruction resets a control bit which then disallows parallel pipeline processing of subsequent mixtures of vector and scalar instructions. In the times-four CP, this means that only scalars can be in execution at a time or only a singular vector at a time. Combinations of vectors and scalars cannot be in execution simultaneously. Refer to the write-up describing FORK and JOIN control for further details on the effect of this instruction.

FORK and JOIN Control

- Purpose

Scalar instructions, operating disjoint from vectors, normally use the four pipes in a parallel fashion. For scalar code of this type, operand and instruction hazard checking hardware is built into the IPU. This hardware prevents the IPU from using scalar operands or instructions that have been modified by prior scalar instructions but which have not yet reached their register or memory destination. However, when vectors and scalars are mixed, the hardware for checking operand or instruction hazards is not effective if these two types are in execution simultaneously. Therefore, the scalar FORK and JOIN instructions have been provided to allow the user to have control over operand and instruction hazard checking between vectors and scalars by either allowing or disallowing their simultaneous execution. In addition to these two scalar instructions, there also exists a means by which the fork indicator can be turned on or off with a VECT or VECTL instruction.

- Setting or Resetting FORK/JOIN Mode

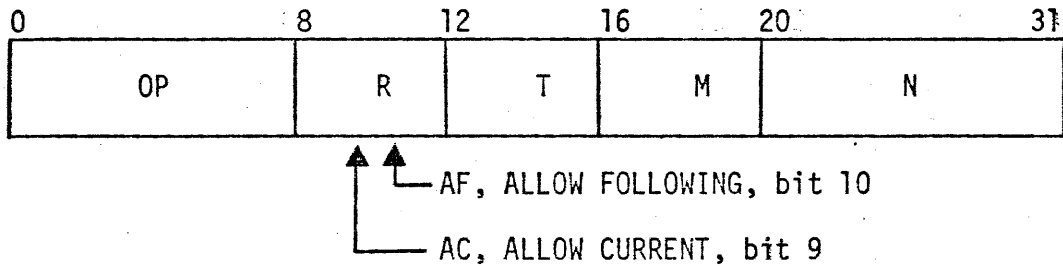
Two methods may be used to set the Fork indicator. One of these is by means of the scalar FORK instruction. The other is by placing a "one" in the ALLOW FOLLOWING (AF) bit of a vector instruction. Resetting of the Fork indicator is done by the JOIN instruction or by a "zero" in the AF bit of a vector instruction.

When the Fork indicator is set, parallel execution of vectors and scalars is allowed by the hardware. When the Fork indicator

is reset, only a singular vector may be in execution in one of the four pipes; or only pure scalar instructions (not mixed with vectors) may be in execution in any of the four pipes.

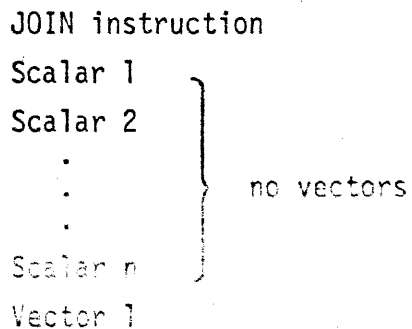
● Uses of the ALLOW CURRENT Bit

The ALLOW CURRENT (AC) bit is contained in the R-field of a vector instruction adjacent to the ALLOW FOLLOWING bit as shown following:



This bit is used in combination with the existing mode of the Fork indicator. The combinations are best described by using some examples.

Example A:



In example A, all scalar instructions are processed in parallel up to Vector 1. Vector 1 will begin processing when all scalar

writes to Central Memory are complete, and no instructions will modify the Vector Parameter File (VPF). This is true no matter what the value of the AC bit is for Vector 1. If the program is to continue in the JOIN mode, the "Allow Following" bit of Vector 1 must be zero.

Example B

```
JOIN instruction
  Scalar 1
  Scalar 2
  .
  .
  Scalar n
  FORK instruction
  Scalar n+1
  Scalar n+2
  Vector 1
```

} no vectors

In example B, all scalar instructions are processed in parallel up to Vector 1. Vector 1 must wait at level 3 of the IPU pipeline until all scalars through "n" have completed their writes to memory, and none of these instructions are of the type that will modify the Vector Parameter File. Any other scalar instruction, either before or after the FORK, can be processed in parallel with Vector 1 if Vector 1 has its "Allow Current" bit set to "one." If the AC bit is "zero," processing of Vector 1 is held up until the preceding conditions are true through Scalar n+2. If this is the case, the FORK instruction had no effect.

● Uses of the ALLOW FOLLOWING Bit

In certain applications it may be desirable for a program to operate in a JOIN mode through Vector 1 but to allow parallel execution of Vectors 1 and 2. This is accomplished with the following program:

Example C

```
JOIN instruction
Scalar 1
Scalar 2 } no vectors
  :
  :
Scalar n }
Vector 1 (AC = 0, AF = 1)
Vector 2 (AC = 1, AF = 0)
      where "0" is a "Don't Care"
```

This program is the same as example A with the exception of the inclusion of Vector 2. The "Allow Following" bit of Vector 1 and the "Allow Current" bit of Vector 2, both being a "one," are the keys that unlock the pipelines for parallel processing.

If either the AF bit of Vector 1 or the AC bit of Vector 2 is "zero" in example C, Vector 2 will proceed to the point of selecting a pipe and initializing that MBU but will not access memory until Vector 1 is complete.

Example D

```
JOIN instruction
Scalar 1
Scalar 2
  :
  :
Scalar n
Vector 1
FORK instruction
Scalar n+1
```

no vectors

Example D is a case where the JOIN mode is desired through Vector 1, and then the FORK instruction is encountered. For this code to be reasonable, the "Allow Following" bit of Vector 1 is "zero." Vector 1 will proceed as described in example A. Vector 1 must complete before any other instruction is processed in the IPU. This includes the FORK, so Scalar n+1 will not be executed until Vector 1 completes.

● AC and AF Summary

The four cases of the AC and AF bits of a vector instruction are summarized as follows:

(AC, AF) = (0, 0)

Finish all the preceding instructions, and then complete this vector before proceeding.

(AC, AF) = (0, 1)

Finish all the preceding instructions, but allow succeeding instructions to proceed while executing this vector.

(AC, AF) = (1, 0)

Begin this vector as soon as all instructions preceding the FORK have been completed, and then finish this vector before proceeding.

(AC, AF) = (1, 1)

Begin this vector as soon as all instructions preceding the FORK have been completed, but allow succeeding instructions to proceed while executing this vector.

The following table relates the AC and AF bits of the R-field to how the instruction string would look without the bits:

WITH BITS	WITHOUT BITS						
<table><tr><td></td><td><u>AC</u></td><td><u>AF</u></td></tr><tr><td>VECTOR</td><td>0</td><td>0</td></tr></table>		<u>AC</u>	<u>AF</u>	VECTOR	0	0	JOIN VECTOR JOIN
	<u>AC</u>	<u>AF</u>					
VECTOR	0	0					
<table><tr><td></td><td><u>AC</u></td><td><u>AF</u></td></tr><tr><td>VECTOR</td><td>0</td><td>1</td></tr></table>		<u>AC</u>	<u>AF</u>	VECTOR	0	1	JOIN VECTOR FORK
	<u>AC</u>	<u>AF</u>					
VECTOR	0	1					
<table><tr><td></td><td><u>AC</u></td><td><u>AF</u></td></tr><tr><td>VECTOR</td><td>1</td><td>0</td></tr></table>		<u>AC</u>	<u>AF</u>	VECTOR	1	0	VECTOR JOIN
	<u>AC</u>	<u>AF</u>					
VECTOR	1	0					
<table><tr><td></td><td><u>AC</u></td><td><u>AF</u></td></tr><tr><td>VECTOR</td><td>1</td><td>1</td></tr></table>		<u>AC</u>	<u>AF</u>	VECTOR	1	1	VECTOR FORK
	<u>AC</u>	<u>AF</u>					
VECTOR	1	1					

Prepare to Branch

PB Mnemonic code

9E Op code

The Prepare-to-Branch instruction is an advisory type instruction to the IPU instruction look-ahead hardware. Execution of a PB instruction does not affect the results of a program in any way; its purpose is to decrease the time taken at a branch instruction in fetching the octet of instructions to which the branch is directed.

The PB instruction develops a β address from its T-, M-, and N-fields in the same way that a standard branch instruction (BCC or BRC) would do if it were placed at the instruction address of the PB instruction. The R-field of the PB instruction should be set to the difference between the instruction address of the PB instruction and the intended branch instruction. This count may not exceed 15 since the R-field is only four bits. Counts of "0" and "1" have special uses.

The internal IPU hardware saves both the β address developed by the PB instruction and the length count specified by the R-field. The length count is decremented by one as each new instruction is entered into the instruction register (IR). At the octet boundary where the look-ahead would normally request the next octet past the octet containing the branch, it recalls the β address saved by the PB instruction and requests it instead of the normal look-ahead octet. In this manner the instruction at the branch address of the target branch instruction will be available for immediate processing following the execution of the target branch instruction.

Should the target branch fail to take the branch, the hardware will realign itself to take the downstream instructions. This is done by

rerequesting the branch instruction's octet if necessary, plus the next octet of look-ahead instructions beyond the branch octet.

Here is an example of PB instruction usage. In this example the R-field is "7," designating seven instruction locations from the PB to the BLB instruction. The branch address developed by the PB is indirect to the PB instruction address, plus eight (Program counter + 8). At this indirect address we find the address of the COSINE routine. The BLB instruction also uses this same indirect address but refers to it via an indirect program counter address plus one. By using the PB instruction in this manner, the first instruction of the COSINE routine will be directly behind the BLB instruction in the IPU pipeline at the completion of BLB execution.

```
PB      7, @ $ + 8
-
-
-
-
-
-
BLB     B1, @ $ + 1
IND     COS
```

Special Extension to PB Instruction

R-field counts of 1 and 0 of the PB instruction have a special use of enabling or disabling the dual branch hardware. If a PB of R-field 1 is executed, the dual branch mechanism is enabled. When enabled, a conditional branch instruction of the type BRC, BCC, or BAE will make a memory request for the octet containing the branch address while waiting at level 3 for the determination of the branch condition.

The octet containing the branch address replaces the look-ahead octet on the assumption that the branch will be taken. If the branch fails, then the octet containing the branch address is discarded; and the normal look-ahead octet is refetched. Dual branch hardware works only if a BRC, BCC, or BAE is positioned in the first four words of an octet.

If a PB of R-field 0 is executed, the dual branch mechanism is disabled. When disabled, nothing is done to fetch an octet along the branch path. The normal look-ahead along the downstream path will continue to function.

Load Arithmetic Exception Mask and Condition Registers

	LEM	Mnemonic code
$(\alpha)_{0-7} \rightarrow AC, AM$	11	Op code

Loads bits 0 through 3 of the contents of location α into the four-bit arithmetic exception condition code register, and loads bits 4 through 7 of the contents of location α into the four-bit arithmetic exception mask register.

Bits 0 through 3 load the arithmetic exception condition code register as follows:

Bit	
0	Divide check
1	Fixed point overflow
2	Floating point overflow
3	Floating point underflow

Bits 4 through 7 load the arithmetic exception mask as follows:

Bit	
4	Divide check
5	Fixed point overflow
6	Floating point overflow
7	Floating point underflow

Result Code

Not set.

Programming Notes

An interrupt signal from the CP to the PPU is activated if an arithmetic exception is detected and the mask bit corresponding to that arithmetic exception has been set to a "one." An interrupt is not possible for that arithmetic exception if the mask bit is set to "zero."

Alteration of the AE condition register and AE mask register by a LEM instruction will cause an arithmetic exception program interruption if the corresponding bits of the AE condition register and AE mask register are both "one" after the LEM instruction has passed through the CP pipeline. Also, a program interruption will occur after completion of a LEM instruction if any of the following pairs of bits from the contents of location α are both "one":

- (0, 4)
- (1, 5)
- (2, 6)
- (3, 7)

This instruction is paired with the BLB and BLX instructions in that the bit positions (bits 0 through 7) agree with the position of the AE condition and AE mask bits stored as a result of a previous BLB or BLX instruction.

<u>Load Arithmetic Register Right</u>	LRL	Mnemonic code
<u>Halfword From Alpha Left Halfword</u>	10	Op code
$(\alpha_h) \rightarrow AR_{rh}$		Arithmetic register file

A halfword operand from memory is entered into the right half of arithmetic register AR. The left half of register AR remains unchanged. The operand selected is from the left half of a central memory or register whole word when not indexed. If indexed, an even index value selects words from the left half of a central memory or register whole word. An odd index value addresses the right halfword.

Result Code: Set arithmetically.

<u>Store Arithmetic Register Right</u>	STRL	Mnemonic code
<u>Halfword into Alpha Left Halfword</u>	26	Op code
$(AR_{rh}) \rightarrow \alpha_h$		

The right half of arithmetic register AR is stored into the left half of a singleword location when not indexed. If indexed, an even

index value selects the left half of a singleword location for storage.
An odd index value addresses the right halfword.

Result Code: Set arithmetically.

Store Clock

SCLK Mnemonic code

CLOCK \rightarrow α

AE Op code

The current value of the 32-bit fixed point CP clock is stored into singleword location α .

Result Code: Set arithmetically.

Vector Select on Not Equal

($SV_{msb} = 1$)

A vector select on not equal instruction generates an output vector \vec{C} composed of elements from vector \vec{B} . The index values given by the elements of vector \vec{A} correspond to the index location of elements that are not selected from vector \vec{B} . All other elements of vector \vec{B} are selected. These selected elements are the ones for which the index values given by vector \vec{A} do not correspond to the index location of elements from vector \vec{B} .

Programming Notes:

(1) Same as (1) under Select on Equal.

(2A) Same as (2A) under Select on Equal. Also, if the last index value plus one is less than the number of elements of vector \vec{B} , then

the self loop length should be set equal to the number of elements of vector \vec{B} .

(2B) If the last index value plus one is equal to the number of elements of vector \vec{B} , then the index list of vector \vec{A} can be examined further to determine if the index values looking backward from this last index value decrease by unity for each step backward. Using this procedure, the value of the last index value found prior to a nonunity decrease of index values can be used for the self loop length specification of the vector select on not equal instruction and still obtain the same result vector \vec{C} . If an index value of zero is reached using this procedure, then no elements of vector \vec{B} will be used by this instruction. The self loop length can be set equal to zero in this case, and no operation will be performed.

(3A) Same as (3A) under Select on Equal.

(3B) Same as (3B) under Select on Equal.

(4) Same as (4) under Select on Equal.

(5) An index list beginning with a value of one, selects the first element of vector $\vec{B}(b_0)$ but not the second element (b_1).

(6) Same as (6) under Select on Equal.

Example: A singleword select on not equal instruction using one self loop of length 8.

<u>Halfword Index Vector \vec{A}</u>	<u>Singleword Vector \vec{B}</u>	<u>Singleword Vector \vec{C}</u>
2, 3	+16,+72,-54,-75	+16,+72,+71,+14
5, 6	+71,-64,-15,+14	
7FFF, -		

Vector Replace on Not Equal

(SV_{msb} = 1)

A vector replace on not equal instruction accepts as inputs a contiguous list of replacement elements from vector \vec{B} and a contiguous list of indices from vector \vec{A} . Elements from vector \vec{B} replace previously existing elements in a central memory region defined as the \vec{C} output array. The index values given by the elements of vector \vec{A} correspond to the index location of elements in the \vec{C} output array that are not replaced. All other elements of vector \vec{C} are replaced. These replaced elements are the ones for which the index values given by vector \vec{A} do not correspond to the index location of elements in vector \vec{C} .

Programming Notes:

(1) The length specification of the self loop (L-field) for a vector replace on not equal instruction should be set equal to the difference between the number of elements in the \vec{C} vector and the number of indices in the \vec{A} vector for a shortened vector operation. Otherwise, the length specification can be set equal to the number of elements in the \vec{C} vector providing that an index boundary limit equal to the largest positive number (7FFF_{hex}) is placed in the data location following the last index value of vector \vec{A} .

(2) Same as (2) under Replace on Equal.

(3) An index list beginning with a value of "one" replaces the first element of vector \vec{C} (C_0) but not the second element (C_1).

(4) Inner or outer loops with vector replace on not equal require a considerable amount of preprocessing on index vector \vec{A} before they can be used.

MEMORANDUM

24 April 1972

TO Dennis Best Bill Beebe
Gary Boswell Gary Cobb
John Gifford Bill Cohagan
Frank Little
Al Riccomi
Tom Treptow

COPY TO Buddy Dean
Charles Stephenson
Joe Watson

FROM Bill Kastner ✓

SUBJECT DECISIONS MADE ON 4X CP CONTEXT SWITCHING

Several decisions were made in a meeting on April 14 relative to context switching on the times four CP. This memo is a confirmation of these decisions.

1. Load, Store, and Exchange Status CCR commands will exist on the 4X CP. The CCR command codes for these operations will not change from those of the 1X CP. They are as follows:

4108 Store Status
4109 Load Status
410A Exchange Status

The data stored in the Status Map consists of six octets of the Register File, the Program Status Doubleword (PSDW), and the 32-bit CP clock. New information stored in the PSDW for the 4X CP is the fork indicator in bit position 21 of the first word of the doubleword. Refer to page 3-26 of the Central Processor Hardware Specification for the PSDW format. Four bits for disabling any one or more of the four pipes during diagnostic tests will be provided in bit positions 12 through 15 of the first word of the Program Status Doubleword.

2. Load, Store, and Exchange Intermediate CCR commands will be changed to Load, Store, and Exchange CP Details CCR commands on the 4X CP. The difference between the old Store intermediate and the new Store CP Details operation will be in the amount of data stored. Store CP Details is essentially the same as the Store Maintenance Details operation on the 4X machine with the difference being the point at which vectors are stopped. With a Store CP Details command the circular address file of the MBU is drained to its empty state; whereas in the case of a Store Maintenance

Details command, the circular address file (CAF) is not emptied since this command is primarily for maintenance use. It is necessary during certain maintenance operations to see the CAF in operation, so the Store Details command is being kept for this purpose.

3. A recent decision, with regard to CCR commands for the CP, is the removal of the Lock and Unlock PC maintenance commands. No satisfactory use has been found for these commands during hardware checkout, so they are being eliminated. Therefore, strike the CCR commands 4102 and 4103 for Unlock PC and Lock PC, respectively, on page 35 of Section G of the ASC System Hardware Description.
4. A CR-bit will be assigned to indicate when a "vector bad guy" is in progress. The CR-bit will continuously monitor the "vector bad guy" state of the CP. A final software check of the CP details map in memory will be made to cover the case in which a bad guy vector was just starting at the time the CR-bit was tested and found to be "zero" or the case in which a bad guy vector had just finished at the time the CR-bit was tested and found to be "one." Appropriate job output messages will be printed by the operating system in the event either of the two cases are detected during operation.
5. Addressability restrictions will be placed on the user when executing a "vector bad guy." In the times four CP the "vector bad guys" are the Vector Select, Replace, and Order instructions. These vectors should not write over their input arrays. This constraint, however, will be the responsibility of the user and will not be protected by hardware.
6. Efforts to reduce the context switching time have dealt with the possibility of using the status map for exchanging CP jobs. The proposed plan is to Store Status when an MCW instruction is encountered or when a stacked MCP occurs. A CP details map will be stored if the CP job is interrupted during execution or if an arithmetic exception, illegal op, spec. error, or protection violation occurs. Interrupting a job during execution is done by issuing a Store CP Details CCR command.

The map stored into memory will contain a bit that indicates whether it was stored as a CP Details map or as a Status map. When a job is returned to CP execution, this bit is transferred to a newly defined CR-bit in the PPU CR-file. This CR-bit is used by the automatic context switch mechanism to determine whether to load a CP Details map or a Status map into the CP when resuming job execution.

Bill Kastner

BILL KASTNER

MEMORANDUM

19 June 1972

TO SE Group Wayne Winkelman (4)
Gary Cobb
Bill Cohagan
Sterling Mathis
Gary Miley
Dave Paterson
Charles Stephenson (4)

FROM Bill Kastner ✓

SUBJECT INSTRUCTION MNEMONIC CODE CHANGES
FOR TIMES FOUR CENTRAL PROCESSOR

The mnemonics for the halfword Load and Store instructions are being changed in the times four Central Processor to a more consistent set. The new halfword Load and Store instructions in the 4X CP are LRL and STRL. The letter sequence used in these mnemonics has been arranged according to the order in which the register operand, then the memory operand is specified in assembler source code. For LRL, the RL sequence indicates the right half register word is loaded from the left half of a central memory word. Similarly for STRL, the RL sequence is an aid in remembering that the right half register word is stored into the left half of a central memory word.

The halfword Load and Store instructions that were available in the times one CP used mnemonics that are not easily associated with their register - memory usage. The following set of mnemonic codes are being used for the 4XCP.

<u>OP CODE</u>	<u>MNEMONIC CODE</u>	<u>DESCRIPTION</u>
15	LLL	Load register left halfword from central memory left halfword
19	LLR	Load register left halfword from central memory right halfword
10	LRL	Load register right halfword from central memory left halfword
1D	LRR	Load register right halfword from central memory right halfword
25	STLL	Store register left halfword into central memory left halfword

Instruction Mnemonic Code
Changes for Times Four
Central Processor

-2-

19 June 1972

<u>OP CODE</u>	<u>MNEMONIC CODE</u>	<u>DESCRIPTION</u>
29	STLR	Store register left halfword into central memory right halfword
26	STRL	Store register right halfword into central memory left halfword
2D	STRR	Store register right halfword into central into central memory right halfword

Bill Kastner

BILL KASTNER

BKfo

June 28, 1972

*Bill
Kastner*

TO: CP Checkout
Hardware Development
Software Development
SDD
System Engineering
System Test

FROM: Bill Kastner

CATEGORY: CT

SUBJECT: Retrofit of ASC #1, #2, and #3 With CP Instructions LRL and STRL

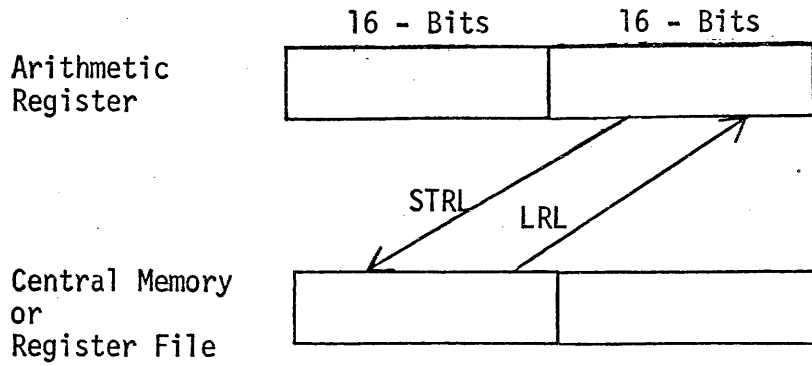
The CP currently has three halfword load and three halfword store instructions. The old and new mnemonic codes for these instructions are:

<u>OP CODE</u>	<u>OLD MNEMONIC</u>	<u>NEW MNEMONIC</u>
15	LH	LLL
19	LL	LLR
1D	LR	LRR
25	STH	STLL
29	STL	STLR
2D	STR	STRR

Four halfword load paths exist from the left or right half memory word to the left or right half register word. A set of four similar paths exist for stores. The remaining halfword load and store path is being included in the times four CP instruction set and will be retrofit on ASC serial numbers 1, 2, and 3. These instructions are:

<u>OP CODE</u>	<u>MNEMONIC CODE</u>	<u>DESCRIPTION</u>
10	LRL	Load Arithmetic Register Right Half from Memory Left Half.
26	STRL	Store Arithmetic Register Right Half into Memory Left Half.

The new instructions provide the following paths from and to the arithmetic register:



A significant feature of this change is the ability to modify the DAI, DCI, DAO, and DCO fields of the vector parameter file with data from the right half of an arithmetic register using the STRL instruction.

Bill Kastner
 Bill Kastner *cms*

MEMORANDUM

4 August 1972

TO Hardware Development
System Engineering
System Planning and Applications

COPY TO Bill Cohagan Dave Paterson
Gary Cobb Charles Stephenson
Sterling Mathis Joe Watson
Sid Nolte Wayne Winkelman (4)

FROM Bill Kastner

SUBJECT NEW INSTRUCTIONS AND NEW FEATURES OF
THE TIMES-FOUR CENTRAL PROCESSOR

New vector instructions for generating and using Boolean vectors have been added to the times-four Central Processor (4XCP), creating the need to re-issue the list of new instructions. This list supersedes the old list of new instructions issued 4 April 1972. In addition to this list of new instructions, all the previous instructions of the times-one CP are implemented on the 4XCP.

Several new features have been added to the 4XCP and are described following the new instructions.

Bill Kastner

BILL KASTNER

WDK:jc

Attachment

Contents

<u>Page</u>	<u>New Scalar Instructions</u>
1	FORK
1	JOIN
2	PB, Prepare to Branch
4	LEM, Load Arithmetic Exception Mask and Condition
6	LRL, Load Arithmetic Register Right Half from Alpha Left Half
7	STRL, Store Arithmetic Register Right Half into Alpha Left Half
8	SCLK, Store 32-bit Fixed Point Clock
 <u>New Vector Instructions</u> 	
9	VMAP, Vector Map Singleword
9	VMAPH, Vector Map Halfword
9	VMAPD, Vector Map Doubleword
14	VSELB, Vector Select Singleword Boolean
14	VSELHB, Vector Select Halfword Boolean
14	VSELDB, Vector Select Doubleword Boolean
16	VREPB, Vector Replace Singleword Boolean
16	VREPHB, Vector Replace Halfword, Boolean
16	VREPDB, Vector Replace Doubleword Boolean
19	VMAPB, Vector Map Singleword Boolean
19	VMAPHB, Vector Map Halfword Boolean
19	VMAPDB, Vector Map Doubleword Boolean

<u>Page</u>	<u>New Vector Instructions</u>
22	VMAX, Vector Max/Min Fixed Point Singleword
22	VMAXH, Vector Max/Min Fixed Point Halfword
22	VMAXF, Vector Max/Min Floating Point Singleword
22	VMAXFD, Vector Max/Min Floating Point Doubleword
24	VCB, Vector Compare Fixed Point Singleword Boolean
24	VCHB, Vector Compare Fixed Point Halfword Boolean
24	VCFB, Vector Compare Floating Point Singleword Boolean
24	VCFDB, Vector Compare Floating Point Doubleword Boolean
26	VCAB, Vector Compare AND Singleword Boolean
26	VCADB, Vector Compare AND Doubleword Boolean
26	VCORB, Vector Compare OR Singleword Boolean
26	VCORDB, Vector Compare OR Doubleword Boolean

New Features

27	Pipe Disable Bits
27	Vector Length Field Specification
30	Dual Look-Ahead

<u>Fork</u>	FORK	Mnemonic Code
1 → Fork Indicator	9A	Op code

The FORK instruction is an advisory type instruction to the IPU control. Execution of the FORK instruction sets the fork indicator bit within the IPU control and allows subsequent vector or scalar instructions to proceed to execution independently. In the times-four CP, this means that any combination of vector or scalar instructions can be in execution simultaneously in each of the four MBU-AU pairs. Refer to the write-up describing FORK and JOIN control for further details on the effect of this instruction.

A FORK instruction with the fork indicator already "on" results in the equivalent of a JOIN followed by a FORK.

<u>Join</u>	JOIN	Mnemonic code
0 → Fork Indicator	9B	Op code

The JOIN instruction is an advisory type instruction to the IPU control. Execution of the JOIN instruction resets a control bit which then disallows parallel pipeline processing of subsequent mixtures of vector and scalar instructions. In the times-four CP, this means that only scalars can be in execution at a time or only a singular vector at a time. Combinations of vectors and scalars cannot be in execution simultaneously. Refer to the write-up describing FORK and JOIN control for further details on the effect of this instruction.

Prepare to Branch

PB	Mnemonic code
9E	Op code

The Prepare-to-Branch instruction is an advisory type instruction to the IPU instruction look-ahead hardware. Execution of a PB instruction does not affect the results of a program in any way; its purpose is to decrease the time taken at a branch instruction in fetching the octet of instructions to which the branch is directed.

The PB instruction develops a β address from its T-, M-, and N-fields in the same way that a standard branch instruction (BCC or BRC) would do if it were placed at the instruction address of the PB instruction. The R-field of the PB instruction should be set to the difference between the instruction address of the PB instruction and the intended branch instruction. This count may not exceed 15 since the R-field is only four bits. Counts of "0" and "1" are not used.

The internal IPU hardware saves both the β address developed by the PB instruction and the length count specified by the R-field. The length count is decremented by one as each new instruction is entered into the instruction register (IR). At the octet boundary where the look-ahead would normally request the next octet past the octet containing the branch, it recalls the β address saved by the PB instruction and requests it instead of the normal look-ahead octet. In this manner the instruction at the branch address of the target branch instruction will be available for immediate processing following the execution of the target branch instruction.

Should the target branch fail to take the branch, the hardware will realign itself to take the downstream instructions. This is done by

rerequesting the branch instruction's octet if necessary, plus the next octet of look-ahead instructions beyond the branch octet.

Here is an example of PB instruction usage. In this example the R-field is "7," designating seven instruction locations from the PB to the BLB instruction. The branch address developed by the PB is indirect to the PB instruction address, plus eight (Program counter + 8). At this indirect address we find the address of the COSINE routine. The BLB instruction also uses this same indirect address but refers to it via an indirect program counter address plus one. By using the PB instruction in this manner, the first instruction of the COSINE routine will be directly behind the BLB instruction in the IPU pipeline at the completion of BLB execution.

PB	7, @ \$ + 8
-	
-	
-	
-	
-	
-	
BLB	.B1, @ \$ + 1
IND	COS

Load Arithmetic Exception Mask and Condition Registers

	LEM	Mnemonic code
$(\alpha)_{0-7} \rightarrow AC, AM$	11	Op code

Loads bits 0 through 3 of the contents of location α into the four-bit arithmetic exception condition code register and loads bits 4 through 7 of the contents of location α into the four-bit arithmetic exception mask register.

Bits 0 through 3 load the arithmetic exception condition code register as follows:

Bit

- 0 Divide check
- 1 Fixed point overflow
- 2 Floating point overflow
- 3 Floating point underflow

Bits 4 through 7 load the arithmetic exception mask as follows:

Bit

- 4 Divide check
- 5 Fixed point overflow
- 6 Floating point overflow
- 7 Floating point underflow

Result Code

Not set.

Programming Notes

An interrupt signal from the CP to the PPU is activated if an arithmetic exception is detected and if the mask bit corresponding to that

arithmetic exception has been set to a "one." An interrupt is not possible for that arithmetic exception if the mask bit is set to "zero."

Alteration of the AE condition register and AE mask register by a LEM instruction will cause an arithmetic exception program interruption if the corresponding bits of the AE condition register and the AE mask register are both "one" after the LEM instruction has passed through the CP pipeline. This implies that a program interruption will occur after completion of a LEM instruction if any of the following pairs of bits from the contents of location α are both "one":

(0, 4)

(1, 5)

(2, 6)

(3, 7)

This instruction is paired with the BLB and BLX instructions in that the bit positions (bits 0 through 7) agree with the position of the AE condition and AE mask bits stored as a result of a previous BLB or BLX instruction.

<u>Load Arithmetic Register Right</u>	LRL	Mnemonic code
<u>Halfword From Alpha Left Halfword</u>	10	Op code

$$(\alpha_h) \rightarrow AR_{rh}$$

A halfword operand from memory is entered into the right half of arithmetic register AR. The left half of register AR remains unchanged. The operand selected is from the left half of a central memory or register whole word when not indexed. If indexed, an even index value selects words from the left half of a central memory or register whole word. An odd index value addresses the right halfword.

Result Code

Set arithmetically.

<u>Store Arithmetic Register Right</u>	STRL	Mnemonic code
<u>Halfword Into Alpha Left Halfword</u>	26	Op code

$$(AR_{rh}) \rightarrow \alpha_h$$

The right half of arithmetic register AR is stored into the left half of a singleword location when not indexed. If indexed, an even index value selects the left half of a singleword location for storage. An odd index value addresses the right halfword.

Result Code

Set arithmetically.

Store Clock

CLOCK → α

SCLK

Mnemonic code

AE

Op code

The current value of the 32-bit, fixed-point CP clock is stored into singleword location α . This clock is incremented by "one" every CP clock pulse. It cycles modulo 2^{32} approximately once every four minutes (based on a 60 ns clock rate).

Result Code

Set arithmetically.

New Vector Instructions

<u>Mnemonic Code</u>	<u>Instruction</u>	<u>Operation Code</u>
VMAP	Vector Map Singlewords	F8
VMAPH	Vector Map Halfwords	F9
VMAPD	Vector Map Doublewords	FB

Vector-Map-on-Equal, SV_{msb} option bit = 0.

A Vector-Map-on-Equal instruction accepts as inputs a contiguous list of indices from vector \vec{A} and a set of source mapping elements from vector \vec{B} . The elements of vector \vec{B} that are mapped are those elements for which the index location in \vec{B} corresponds to the index value given by the elements of vector \vec{A} . Elements from source mapping vector \vec{B} replace previously existing elements in a central memory region defined as the \vec{C} output vector. Elements of the \vec{C} output vector that are replaced are those elements for which the index location in \vec{C} corresponds to the index value given by the elements of vector \vec{A} . Elements of vector \vec{B} do not replace elements of vector \vec{C} for those index locations in \vec{C} that are not represented in the index list given by vector \vec{A} .

This instruction differs from the Vector Replace instruction in the manner in which elements of vector \vec{B} are used. Vector Replace uses consecutive elements of vector \vec{B} , whereas Vector Map uses only those elements of Vector \vec{B} that are mapped by the specification of vector \vec{A} .

Programming Notes

For self-loops:

- 1A) The length specification of the self-loop (L-field) for a Vector-Map-on-Equal instruction should be set equal to the

number of elements of a self-loop of the \vec{C} vector. Or, if the \vec{B} vector is the greater in length, then set the L-field equal to the number of elements of vector \vec{B} .

- 1B) It is possible to shorten the vector operation and still obtain the same result vector \vec{C} by setting the self-loop length equal to one plus the value of the last index in vector \vec{A} .
- 2A) If the vector length is specified according to 1A above, then an index boundary limit equal to the largest positive number (7FFF hex) must be placed in the data location following the last index value of vector \vec{A} .
- 2B) If the vector length is specified according to 1B above, then the index boundary limit is not necessary.
- 3) Each index value given by vector \vec{A} is a positive, fixed-point halfword. Vector \vec{A} should be a contiguous list of monotone increasing halfwords.
- 4) An index value of zero maps the first element of vector \vec{B} into the first element of vector \vec{C} .

Example 1: A singleword Vector-Map-on-Equal ($SV_{msb}=0$) instruction using a self-loop of length 8.

<u>Singleword Index Vector \vec{A}</u>	<u>Singleword Vector \vec{B}</u>	<u>Singleword Vector \vec{C} Before Replacement</u>	<u>Singleword Vector \vec{C} After Replacement</u>
2, 3	-14	16	16
5, 6	-70	82	82
7FFF, -	-25	27	-25
	-34	36	-34
	-69	71	71
	-30	32	-30
	- 6	8	- 6
	-12	14	14

Vector-Map-on-Not-Equal, SV_{msb} option bit = 1

A Vector-Map-on-Not-Equal instruction accepts as inputs a contiguous list of indices from vector \vec{A} and a set of source mapping elements from vector \vec{B} . The elements of vector \vec{B} that are mapped are those elements for which the index location in \vec{B} is not represented in the index list given by vector \vec{A} . Elements from source mapping vector \vec{B} replace previously existing elements in a central memory region defined as the \vec{C} output vector. Elements of the \vec{C} output vector that are replaced are those elements for which the index location in \vec{C} is not represented in the index list given by vector \vec{A} . Elements of vector \vec{B} do not replace elements of vector \vec{C} for those index locations that correspond to the index value given by the elements of vector \vec{A} .

Programming Notes

For self-loops:

- (1) The length specification of the self-loop (L-field) for a Vector-Map-on-Not-Equal instruction should be set equal to the number of elements of a self-loop of the \vec{C} vector. Or, if the \vec{B} vector is the greater in length, then set the L-field equal to the number of elements of vector \vec{B} .
- (2) An index boundary limit equal to the largest positive number (7FFF hex) must be placed in the data location following the last index value of vector \vec{A} .
- (3) Each index value given by vector \vec{A} is a positive, fixed-point halfword. Vector \vec{A} should be a contiguous list of monotone increasing halfwords.

- (4) An index list beginning with a value of "one" maps the first elements of vector \vec{B} into \vec{C} but not the second element (element C_0 is replaced with B_0 but not C_1 by B_1).

Example 2: A singleword Vector-Map-on-Not-Equal ($SV_{msb}=1$) instruction using a self-loop of length 8.

<u>Singleword Index Vector \vec{A}</u>	<u>Singleword Vector \vec{B}</u>	<u>Singleword Vector \vec{C} Before Replacement</u>	<u>Singleword Vector \vec{C} After Replacement</u>
2, 3	-14	16	-14
5, 6	-70	82	-70
7FFF, -	-25	27	27
	-34	36	36
	-69	71	-69
	-30	32	32
	- 6	8	8
	-12	14	-12

<u>Mnemonic Code</u>	<u>Instruction</u>	<u>Operation Code</u>
VSELB	Vector Select Singleword Boolean	B4
VSELHB	Vector Select Halfword Boolean	B5
VSELDB	Vector Select Doubleword Boolean	B7

Vector-Select-on-One, SV_{msb} option bit = 0

A Vector-Select-on-One instruction generates an output vector \vec{C} composed of elements from vector \vec{B} . The elements selected from vector \vec{B} are those for which the location in vector \vec{B} corresponds to the location of nonzero elements of vector \vec{A} . Selected elements are stored into contiguous locations of vector \vec{C} .

Vector-Select-on-Zero, SV_{msb} option bit = 1

A Vector-Select-on-Zero instruction generates an output vector \vec{C} composed of elements from vector \vec{B} . The elements selected from vector \vec{B} are those for which the location in vector \vec{B} corresponds to the location of zero elements of vector \vec{A} . Selected elements are stored into contiguous locations of vector \vec{C} .

Programming Notes

For self-loops:

- (1) The length specification of the self-loop (L-field) for a Vector-Select-Boolean instruction is set equal to the number of elements of vector \vec{A} or \vec{B} .
- (2) Each element of vector \vec{A} is a halfword that assumes one of two Boolean values. "Zero" is assumed if the value is zero, and "one" is assumed if the value is nonzero.

Example 1: A singleword Vector-Select-on-One ($SV_{msb}=0$) instruction using a self-loop of length 8.

<u>Halfword Boolean Vector \vec{A}</u>	<u>Singleword Vector \vec{B}</u>	<u>Singleword Selected Vector \vec{C}</u>
0, 0	+16	-54
1, 1	+82	-75
0, 1	-54	-64
1, 0	-75	-15
	+71	
	-64	
	-15	
	+14	

Example 2: A singleword Vector-Select-on-Zero ($SV_{msb}=1$) instruction using a self-loop of length 8.

<u>Halfword Boolean Vector \vec{A}</u>	<u>Singleword Vector \vec{B}</u>	<u>Singleword Selected Vector \vec{C}</u>
0, 0	+16	+16
1, 1	+82	+82
0, 1	-54	+71
1, 0	-75	+14
	+71	
	-64	
	-15	
	+14	

<u>Mnemonic Code</u>	<u>Instruction</u>	<u>Operation Code</u>
VREPB	Vector Replace Singleword Boolean	BC
VREPHB	Vector Replace Halfword Boolean	BD
VREPDB	Vector Replace Doubleword Boolean	BF

Vector-Replace-on-One, SV_{msb} option bit = 0

A Vector-Replace-on-One instruction accepts as inputs a continuous list of replacement elements from vector \vec{B} and a continuous list of Boolean elements from vector \vec{A} . Elements from vector \vec{B} replace previously existing elements in a central memory region defined as the \vec{C} output vector. Elements of the \vec{C} output vector that are replaced with elements of vector \vec{B} are those elements for which the location in the \vec{C} output vector corresponds to the location of nonzero elements of vector \vec{A} . Elements of the \vec{C} output vector remain unchanged in those locations for which the corresponding location in vector \vec{A} is zero.

Vector-Replace-on-Zero, SV_{msb} option bit = 1

A Vector-Replace-on-Zero instruction accepts as inputs a continuous list of replacement elements from vector \vec{B} and a continuous list of Boolean elements from vector \vec{A} . Elements from vector \vec{B} replace previously existing elements in a central memory region defined as the \vec{C} output vector. Elements of the \vec{C} output vector that are replaced with elements of vector \vec{B} are those elements for which the location in the \vec{C} output vector corresponds to the location of zero elements of vector \vec{A} . Elements of the \vec{C} output vector remain unchanged in those locations for which the corresponding location in vector \vec{A} is nonzero.

Programming Notes

For self-loops:

- (1) The length specification of the self-loop (L-field) for a Vector-Replace-Boolean instruction is set equal to the number of elements of vector \vec{A} or \vec{C} .
- (2) Each element of vector \vec{A} is a halfword that assumes one of two Boolean values. "Zero" is assumed if the value is zero, and "one" is assumed if the value is nonzero.

Example 1: A singleword Vector-Replace-on-One ($SV_{msb}=0$) instruction using a self-loop of length 8.

<u>Halfword Boolean Vector \vec{A}</u>	<u>Singleword Vector \vec{B}</u>	<u>Singleword Vector \vec{C} Before Replacement</u>	<u>Singleword Vector \vec{C} After Replacement</u>
0, 0	-54	16	16
1, 1	-72	82	82
0, 1	-64	27	-54
1, 0	-15	36	-72
		71	71
		32	-64
		8	-15
		14	14

Example 2: A singleword Vector Replace on Zero ($SV_{msb}=1$) instruction using a self-loop length of 8. (Following page.)

<u>Halfword Boolean Vector A</u>	<u>Singleword Vector B</u>	<u>Singleword Vector C Before Replacement</u>	<u>Singleword Vector C After Replacement</u>
0, 0	-54	16	-54
1, 1	-72	82	-72
0, 1	-64	27	27
1, 0	-15	36	36
		71	-64
		32	32
		8	8
		14	-15

<u>Mnemonic Code</u>	<u>Instruction</u>	<u>Operation Code</u>
VMAPB	Vector Map Singleword Boolean	FC
VMAPHB	Vector Map Halfword Boolean	FD
VMAPDB	Vector Map Doubleword Boolean	FF

Vector-Map-on-One, SV_{msb} option bit = 0

A Vector-Map-on-One instruction accepts as inputs a continuous list of Boolean elements from vector \vec{A} and a set of source mapping elements from vector \vec{B} . The elements of vector \vec{B} that are mapped are those elements for which the location in \vec{B} corresponds to the location of nonzero elements of vector \vec{A} . Elements from source mapping vector \vec{B} replace previously existing elements in a central memory region defined as the \vec{C} output vector.

Elements of the \vec{C} output vector that are replaced are those elements for which the location in \vec{C} corresponds to the location of nonzero elements of vector \vec{A} . Elements of the \vec{C} output vector remain unchanged in those locations for which the corresponding location in vector \vec{A} is zero.

Vector-Map-on-Zero, SV_{msb} option bit = 1

A Vector-Map-on-Zero instruction accepts as inputs a continuous list of Boolean elements from vector \vec{A} and a set of source mapping elements from vector \vec{B} . The elements of vector \vec{B} that are mapped are those elements for which the location in \vec{B} corresponds to the location of zero elements of vector \vec{A} . Elements from source mapping vector \vec{B} replace previously existing elements in a central memory region defined as the \vec{C} output vector.

Elements of the \vec{C} output vector that are replaced are those elements for which the location in \vec{C} corresponds to the location of zero elements of vector \vec{A} . Elements of the \vec{C} output vector remain unchanged in those locations for which the corresponding location in vector \vec{A} is nonzero.

Programming Notes

For self-loops:

- (1) The length specification of the self-loop (L-field) for a Vector-Map-Boolean instruction is set equal to the number of elements of vector \vec{A} . Vectors \vec{B} and \vec{C} should be of this same length.
- (2) Each element of vector \vec{A} is a halfword that assumes one of two Boolean values. "Zero" is assumed if the value is zero, and "one" is assumed if the value is nonzero.

Example 1: A singleword Vector-Map-on-One ($SV_{msb}=0$) instruction using a self-loop of length 8.

<u>Halfword Boolean Vector \vec{A}</u>	<u>Singleword Vector \vec{B}</u>	<u>Singleword Vector \vec{C} Before Mapping</u>	<u>Singleword Vector \vec{C} After Mapping</u>
0, 0	-54	16	16
1, 1	-72	82	82
0, 1	-64	27	-64
1, 0	-15	36	-15
	-29	71	71
	- 5	32	- 5
	-47	8	-47
	- 2	14	14

Example 2: A singleword Vector-Map-on-Zero ($SV_{msb}=1$) instruction using a self-loop of length 8.

<u>Halfword Boolean Vector \vec{A}</u>	<u>Singleword Vector \vec{B}</u>	<u>Singleword Vector \vec{C} Before Mapping</u>	<u>Singleword Vector \vec{C} After Mapping</u>
0, 0	-54	16	-54
1, 1	-72	82	-72
0, 1	-64	27	27
1, 0	-15	36	36
	-29	71	-29
	- 5	32	32
	-47	8	8
	- 2	14	- 2

<u>Mnemonic Code</u>	<u>Instruction</u>	<u>Operation Code</u>
VMAX	Vector Max/Min Fixed-Point Singleword	F4
VMAXH	Vector Max/Min Fixed-Point Halfword	F5
VMAXF	Vector Max/Min Floating-Point Singleword	F6
VMAXD	Vector Max/Min Floating-Point Doubleword	F7

Vector Maximum, SV_{msb} option bit = 0

A Vector-Maximum instruction forms an output vector \vec{C} composed of the larger of the elements from either vector \vec{A} or vector \vec{B} . That is, element c_i assumes the larger arithmetic value of the elements a_i or b_i .

$$c_i = \text{MAX}(a_i, b_i)$$

Vector Minimum, SV_{msb} option bit = 1

A Vector-Minimum instruction forms an output vector \vec{C} composed of the smaller of the elements from either vector \vec{A} or vector \vec{B} . That is, element c_i assumes the smaller arithmetic value of the elements a_i or b_i .

$$c_i = \text{MIN}(a_i, b_i)$$

Example 1: A Fixed-Point, Singleword-Vector-Maximum instruction with a self-loop of length 8.

<u>Vector \vec{A}</u>	<u>Vector \vec{B}</u>	<u>Vector \vec{C}</u>
40	72	72
75	20	75
-11	45	45

<u>Vector \vec{A}</u>	<u>Vector \vec{B}</u>	<u>Vector \vec{C}</u>
56	56	56
32	- 9	32
16	64	64
97	28	97
21	20	21

Example 2: A Fixed-Point, Singleword-Vector-Minimum instruction with a self-loop length of 8.

<u>Vector \vec{A}</u>	<u>Vector \vec{B}</u>	<u>Vector \vec{C}</u>
40	72	40
75	20	20
-11	45	-11
56	56	56
32	- 9	- 9
16	64	16
97	28	28
21	20	20

<u>Mnemonic Code</u>	<u>Instruction</u>	<u>Operation Code</u>
VCB	Vector Compare Fixed-Point Singleword Boolean	F0
VCHB	Vector Compare Fixed-Point Halfword Boolean	F1
VCFB	Vector Compare Floating-Point Singleword Boolean	F2
VCFDB	Vector Compare Floating-Point Doubleword Boolean	F3

All ALCT options of the arithmetic compare instructions on page 3-200 can be used to generate Boolean vector outputs. A Boolean vector is a vector containing elements having a value of either "zero" or "one." For the arithmetic compare instructions, a "one" is placed in the \vec{C} output vector in each halfword location corresponding to the location of true comparisons of elements in the input vectors \vec{A} and \vec{B} . A "zero" is placed in the halfword location corresponding to the location of false comparisons of the \vec{A} and \vec{B} input vectors.

No item count is stored for any of the Boolean vector compare instructions.

Example 1: A Vector Compare Fixed-Point Singleword Boolean instruction with ALCT comparison option set to search for "greater than or equal to."

<u>Singleword Vector \vec{A}</u>	<u>Singleword Vector \vec{B}</u>	<u>Halfword Vector \vec{C}</u>
40	72	0
75	20	1
-11	45	0
56	56	1
32	- 9	1
16	64	0

Singleword
Vector \vec{A}

97

21

Singleword
Vector \vec{B}

28

20

Halfword
Vector \vec{C}

1

1

<u>Mnemonic Code</u>	<u>Instruction</u>	<u>Operation Code</u>
VCAB	Vector Compare AND, Singleword Boolean	EA
VCADB	Vector Compare AND, Doubleword Boolean	EB
VCORB	Vector Compare OR, Singleword Boolean	EE
VCORDB	Vector Compare OR, Doubleword Boolean	EF

All ALCT options of the logical compare instructions on page 3-203 can be used to generate Boolean vector outputs. A Boolean vector is a vector containing elements having a value of either "zero" or "one." For the logical compare instructions, a "one" is placed in the \vec{C} output vector in each halfword location corresponding to the location of true comparisons of elements in the input vectors \vec{A} and \vec{B} . A "zero" is placed in the halfword location corresponding to the location of false comparisons of the \vec{A} and \vec{B} input vectors.

No item count is stored for any of the Boolean vector compare instructions.

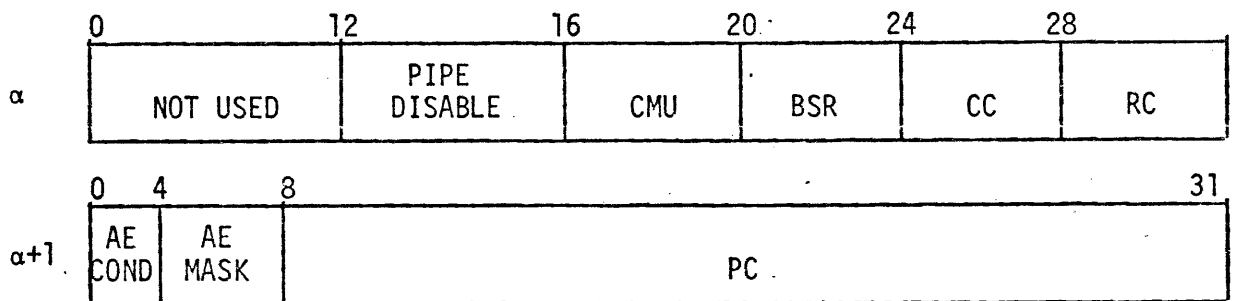
Example 1: A Vector Compare OR Singleword Boolean instruction with ALCT comparison option set to search for "mixed zeros and ones."

<u>Singleword Vector \vec{A}</u>	<u>Singleword Vector \vec{B}</u>	<u>Halfword Vector \vec{C}</u>
005A	0000	01
0000	0000	00
0048	0024	01
5A5A	A5A5	00

New Features

Pipe Disable Bits

The times-four CP is provided with four bits in the Program Status Doubleword (PSDW) for individually disabling any combination of the four parallel pipelines. Bit positions 12, 13, 14, and 15 of the first word of the PSDW disables pipes 0, 1, 2, and 3, respectively, when the bits are "one." A "zero" in these bit positions enables the pipes if they exist in the system. The Program Status Doubleword information is formatted as shown following. A description of the various fields is given in the 1X CP Hardware Specifications on pages 3-19 through 3-27.



The pipe disable bits are primarily intended for use by maintenance and diagnostic testing. As an example of a diagnostic application, it is possible to disable all pipes, except pipe 0, and run an AU or MBU diagnostics program. Then, pipes 0, 2, and 3 can be disabled and pipe 1 enabled and the AU or MBU tests repeated. In this manner, all four pipes can be tested, one at a time. If a particular pipe fails the test, then the operating system can be informed of the malfunctioning pipe. The operating system can be instructed to turn off the pipe in which the failure was found by setting the proper disable bit in the PSDW of subsequent jobs as they are assigned to the CP. The CP can continue processing

jobs in this degraded mode until the failure can be corrected. Also, if the complete system files and status could be saved at periodic checkpoints following successful diagnostic tests, then it would be possible to rerun all jobs executed since the last good checkpoint prior to the detection of a failure by the diagnostic test program.

Vector Length Field Specification

This variation in hardware design existing between the times-one IPU and the times-four IPU has to do with a value of zero in the NI- or NØ-fields of the vector parameter file. In the times-one IPU, if NI is zero, the self-loop routine is executed once, there is no inner loop, and the outer loop count (NØ) is not examined. Also, if NØ is zero, the specified vector operation is executed NI times; and, then, the operation is terminated. This is no longer true in the 4X CP.

In the 4X CP, the vector operation becomes a NO OPERATION if any of the L-, NI-, or NØ-fields are zero. In fact, the MBU will not even be initialized with data from the vector parameter file. The IPU4 detects the zero field condition and terminates the vector before it has had a chance to start initialization.

Dual Look-Ahead

A dual look-ahead procedure is implemented in the 4X CP hardware for decreasing the wait time for acquiring instructions at the branch address of a conditional branch instruction. In this method, two instruction buffers (KA and KB) hold instructions to be executed. Each buffer contains eight instructions. These buffers operate in a toggling fashion such that one contains the octet of instructions from which the current instruction is being read, while the other octet contains the look-ahead octet (eight words) of instructions. These roles are reversed when the address of the current instruction moves across the octet boundary into the octet of instructions that were fetched by the look-ahead hardware.

A branch instruction breaks the normal flow of instructions through the instruction processing pipeline when the branch is taken. A delay in addition to the wait time for acquiring instructions along the branch path is due to the time taken in waiting for the branch condition to be determined. For example, a Branch-on-Result-Code instruction must wait for the result code to be set by the last result code modifying instruction prior to the branch.

The dual look-ahead hardware prefetches the address of the branch instruction before it is known whether the branch will be taken. However, the request for the branch address is made only if the conditional branch instruction is located in one of the first four words of an octet.

Imposing this restriction on the location of the branch instruction is for the purpose of instruction recovery in cases where the branch is not taken. In cases where the branch fails, there will be four instruc-

tions remaining in the current instruction buffer which provide work for the instruction preprocessor that can be overlapped with the request to recover instructions along the nonbranch path.

In cases where the branch is taken, the instructions along the branch path will be available for execution earlier than if the dual look-ahead hardware were not used. This is because the time required to fetch the instructions at the branch address can be overlapped with the determination of the branch condition.

Bill Kastner

TO: CP Checkout
Hardware Development
Software Development
SDD
System Engineering
System Test

FROM: ✓ Bill Kastner

CATEGORY: CT

SUBJECT: CP Instruction Clarifications and Instruction Timing Changes

The clarifications and changes of this bulletin are to be made in the Volume titled "The ASC System Central Processor - May 1971". The clarifications involve (1) the limits that must be observed when using large numbers in the BCLE and BCG instructions and (2) the scale factor size when using the conversion instructions. The changes are the latest figures for divide times in the Arithmetic Unit.

A. The BCG and BCLE instructions (operation codes 84, 85, 86, and 87) require clarification concerning the range of numbers over which these instructions are effective. Programming Notes are being added to these four instructions as follows:

- 1) BCLE, Op. code 84 and
BCG, Op code 85

Programming Notes: This instruction is effective for numbers within the range $|(AR) + (AT) - (AT+1)| \leq 2^{31}-1$.

Also, neither indexed nor indirect branch addressing is possible for BCLE or BCG instructions.

- 2) BCLE, Op. code 86 and
BCG, Op code 87

Programming Notes: This instruction is effective for numbers within the range $|(XR) + (AT) - (AT+1)| \leq 2^{31}-1$.

Also, neither indexed nor indirect branch addressing is possible for BCLE or BCG instructions.

These Programming Notes will appear under the BCLE, BCG instructions on pages 3-135 through 3-138 of the Volume "The ASC System - Central Processor - May 1971" when it is re-issued.

- B. A clarification of the size of scale factor is needed for both scalar and vector conversion instructions. The instructions covered by this clarification are:

Floating Point to Fixed Point Conversion

Scalar	Vector
FLFX	VFLFX
FLFH	VFLFH
FDFX	VDFX

Fixed Point to Floating Point Conversion

Scalar	Vector
FXFL	VFXFL
FXFD	VFXFD
FHFL	VHFL
FHFD	VHFD

For these instructions, the scale factor is supplied as one of the arguments for the conversion process and is obtained from halfword location α_h . The scale factor is a 9-bit signed integer and is represented in 2's complement notation for negative numbers. The sign bit is located in bit position 7 (counting 0 through 15) of the 16-bit halfword.

- C. Timing changes have occurred, particularly in the scalar and vector divide operations. The Arithmetic Unit time for division in CP clock times is as follows:

<u>Scalar Divide</u>		<u>Vector Divide</u>	
D	30	VD	18
DI	30	VDH	18
DH	30	VDF	8
DIH	30	VDFD	18
DF	15		
DFD	26		

Changes to Table I
on page 2-4.

Changes to Table 3
on page 2-22.

Timing changes have also occurred in the scalar Increment (Decrement) and Skip instructions and in the Stack Modify instruction. These changes are to be made in Table 1 on page 2-6.

	was	is now
ISE	3	4
ISNE	3	4
DSE	3	4
DSNE	3	4
MOD	3**	5**

** Modify takes 2 passes through the CP pipeline from level 3.

Bill Kastner

Bill Kastner

MEMORANDUM

25 August 1972

TO Al Riccomi

COPY TO Gary Boswell
Charles Stephenson
Joe Watson

FROM Bill Kastner

SUBJECT PROPOSED VIRTUAL MEMORY/DEMAND PAGING
HARDWARE DESIGN FOR ASC

When an instruction request is made outside the page boundary (page fault), the following is possible with changes:

1. The protection registers for read, write, and execute are used to define page boundaries. If more than one page is defined, then additional pages must be defined over consecutive virtual addresses.
2. A request for an address outside the resident page boundary results in a protection violation (PV) associated with the octet of instructions requested for the KA or KB files.
3. When an instruction with a PV flag reaches level 3, the PV signal is sent to Master Hard Core causing a CP Details exchange. This is different than the present design in that a PV results in a CP maintenance exchange which is more abrupt, and the job is not restartable.
4. Following the exchange, the operating system must examine the CP details map and determine the cause of the protection violation - whether it is caused by:

	<u>Register</u>
(a) Instruction request	(P3, LA)
(b) Load File	(AR)
(c) Store File	(AR)
(d) Indirect request	(AR)
(e) Scalar read	XBA
(f) Scalar store	ZBA

	<u>Register</u>
(g) Vector read	-
(h) Vector store	ZBA

5. Upon determining the cause by examining the various registers listed in the preceding, the operating system can load the non-resident page into real memory and then reschedule the CP job for execution.

If the request is made by a Load File, Store File, or Indirect instruction, then the page fault address can be found in the AR register of the CP details map. The control register (C3) at level 3 would have to be examined to determine whether AR contained an address of a LF, LFM, ST, SFM, or Indirect instruction before it would be worthwhile to check AR for its page fault address.

Branch instructions cause an instruction request, but the branch address in AR is moved to LA when the request is made; so it would be easier to look to LA for an address that is not in real memory for branches.

Operand requests by scalars are sent through the XBA register of the MBU, so read addresses would be found in the XBA location of the CP details map.

Scalar Stores are more difficult to implement. The Hard Core of the MBU would have to be changed to cause a final store of data still residing in the ZB-buffer. The forced store must take place because the MCU does not accept data for which an address protection violation occurs. The operating system would then have to extract the ZB octet from the CP details map and insert it into the virtual page space where it was to be stored but could not be stored because of the page fault. Vector stores to memory could be handled in an identical way, obtaining the storage address in each case from ZBA of the CP details map.

Vector reads are the most difficult of all because the page fault address has been discarded by the CP. The look-ahead hardware continues past the PV-causing address and does not save requested addresses. Also, the vector parameter file (VPF) that initialized the vector has been overwritten by a possible next vector instruction. Losing the VPF means that the pattern of addressing memory cannot be reconstructed by the operating system.

A possible solution exists to this problem. That is, halting the CP for a PV, then sending three CCR commands to the MBU, requesting the hard core register 0A containing the PV address. Now, a CCR command is sent to the CP requesting the CP details exchange. All of the process of examining the different registers and control information in the CP details map would still have to be performed to determine whether

the CP was running a vector which caused the page fault. If a vector read caused the fault, then the CCR-extracted address would be used to call in the new page.

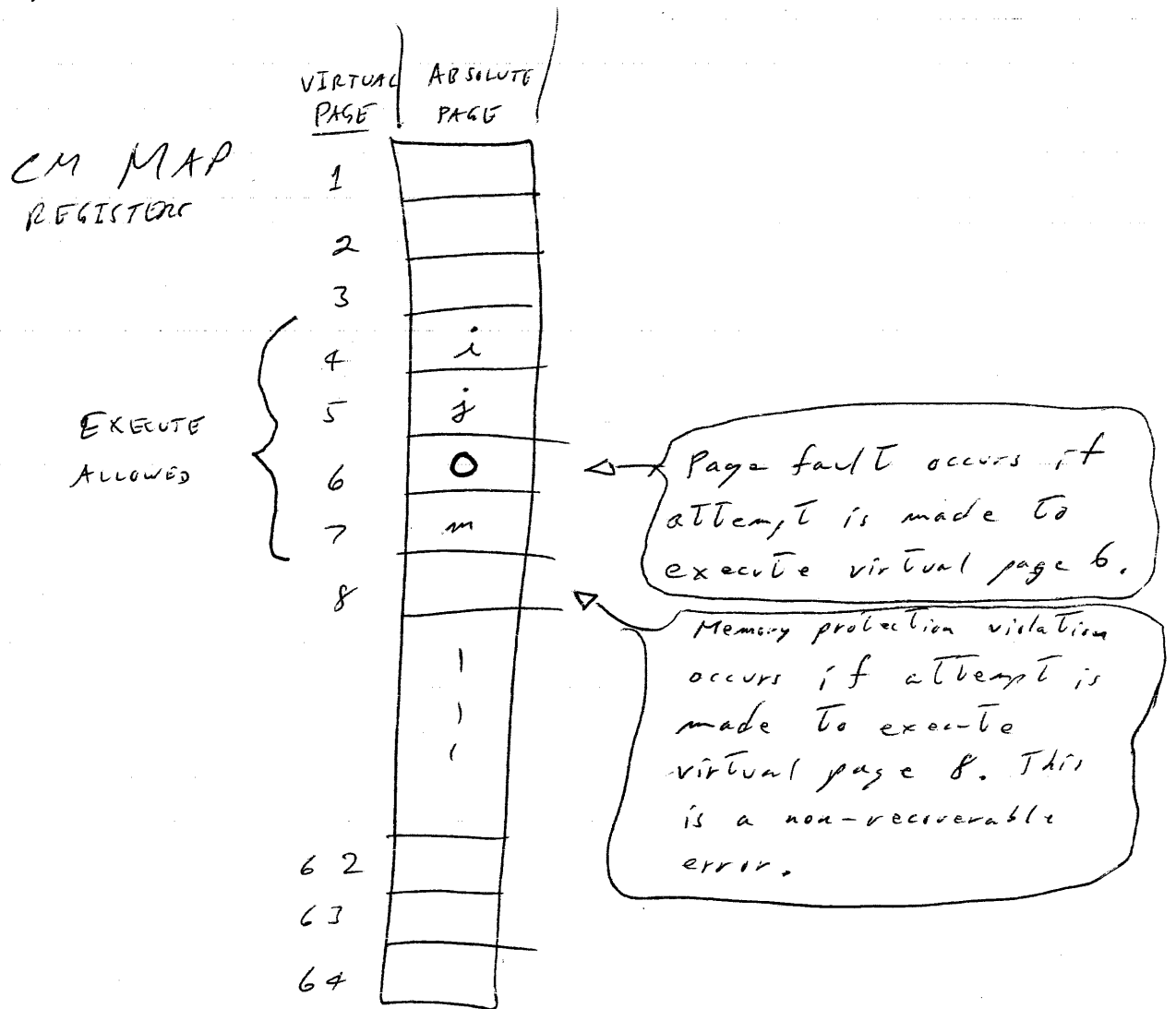
The process proposed is far from being "clean" or efficient; it is only a possible solution to the Virtual Memory/deman paging problem.

BILL KASTNER

WDK:jc

Bill,

1. Page faults will occur within the region defined as execute allowed by the protection register pairs. I. E.:



2. Have you considered any MCU changes to make a "cleaner" solution for some problems? *al*

MEMORANDUM

20 September 1972

TO All ASC Personnel
FROM Bill Kastner
SUBJECT VECTOR INNER OR OUTER LOOP LENGTH
SPECIFICATION CHANGE

Assembly Language programmers should take special note of the vector length specification change on the times four CP and subsequent ASC's using the 4X CP Instruction Processing Unit. This change requires that assembly code previously containing values of "zero" for the inner or outer loop counts must now contain the value "one." If a count of "zero" is placed in any one or more of the L, NI, or NO fields of the vector parameter file, then the vector becomes a NOP.

BILL KASTNER

Bill Kastner

BKfo

MEMORANDUM

20 December 1972

TO Hardware Development
System Planning and Applications
4X CP Test Development

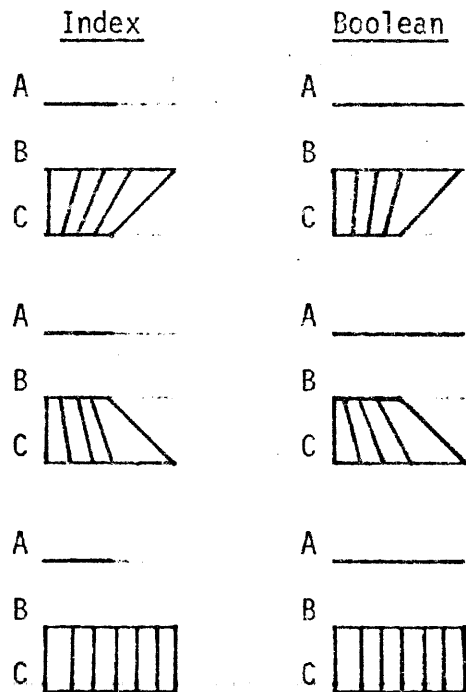
COPY TO Bill Cohagan
Tom Courtney
Dave Paterson
Joe Watson
Wayne Winkelman

FROM → Bill Kastner

SUBJECT MULTILoop BEHAVIOR OF SELECT,
REPLACE, AND MAP INSTRUCTIONS

Clarification of the Select, Replace, and Map instructions is needed, particularly for multiple loop operations. First, in order to bring the varying lengths of these vectors to light, observe the table and figure following representing the vector lengths for one self-loop.

Inst.	Vector Self-Loop Length	
	Index	Boolean
Select A	Short	Long
B	Long	Long
C	Short	Short
Replace A	Short	Long
B	Short	Short
C	Long	Long
Map A	Short	Long
B	Long	Long
C	Long	Long



The figure shows six cases - two for each of the Select, Replace, and Map instructions. The index vector is the \vec{A} vector and is normally the short vector for the three index cases. Boolean vectors use a long \vec{A} vector for the Boolean operators. These Boolean vectors are as long as the \vec{B} vector for Select and Map and as long as the \vec{C} vector for Replace and Map. The \vec{C} vector is short for Select, while the \vec{B} vector is short for Replace for both index and Boolean operators.

The vector self-loop length, L , for Select, Replace, and Map is based on the long vector length. For either index or Boolean operations, the long vector is \vec{B} for Select and Map and \vec{C} for Replace and Map.

A key hardware concept is that vector address generation is independent of vector element usage. That is, the Memory Buffer Unit (MBU) develops vector addresses and counts down its vector length register some interval of time ahead of vector element usage by the Arithmetic Unit (AU). For these vectors, the AU does not know of the end of a self-loop. The AU simply continues processing data as it appears at the AU receiver register.

The application of inner or outer loop deltas to the \vec{A} and \vec{B} vector addresses occur after L (self-loop length) element addresses have been generated, regardless of the number of elements that have been used up to this point in the vector operation. The only way an inner or outer loop delta can be applied at the proper end of self-loop of a short vector is by finding a problem for which the number of indices per self-loop is a constant; and this constant times another multiplier, K , is equal to L (the long vector length). Even for this rare case, the deltas are applied to the \vec{A} vector only once every K times that the data vector length L is reached. If this unlikely condition is not satisfied, then the inner or outer loop delta will be applied somewhere in the middle of a self-loop of the short vector. Therefore, if multiple self-loops are used, then it is safer to use a delta value of one (1) for inner or outer loop increments in order to avoid having the index vector jump some non-unity distance out in the middle of a self-loop.

To summarize the use of loop increments, all short vectors listed in the table preceding should use inner and outer delta increments that are +1. Any long vector in the table may use arbitrary increments for inner or outer loop deltas.

Another important fact to remember when using the indexed Select, Replace, and Map instruction is that the AU accumulator is not reset to "zero" at the end of each self-loop. The AU accumulator is compared against the index value presented by vector \vec{A} to determine which elements are to be transferred by the vector operation. Therefore, in order to make these multiloop vectors work, the index values presented to the AU on

the \bar{A} vector side should continue to increase in value from one self-loop to the next. Such index vectors are generated by the Vector Compare and Peak Pick instructions with the VI most significant bit set to "one." The indexed Select, Replace, and Map instructions do not employ the VI most significant bit option but do act as though this option bit was set to "one."

Bill Kastner

BILL KASTNER

WDK:jc

pg - 7

MEMORANDUM

9 April 1973

TO System Planning Group

COPY TO Gary Boswell Al Ricconi
Gary Cobb Charles Stephenson
Bill Cohagan Marvin Talbott
Sid Nolte Hollie Thompson

FROM Bill Kastner

SUBJECT TIMES-FOUR INSTRUCTION PROCESSING UNIT
SECTIONAL DESCRIPTION

A general description of the Times-Four Central Processor is included as an attachment to this memo. This material was originally written for, and has since been incorporated in, the Central Processor Volume of the ASC System Hardware Manual. This material may be found beginning on page 1-14 of this manual. It is released here in memo form for ease of access.

Bill Kastner

BILL KASTNER

WDK:jsc

Attachment

GENERAL DESCRIPTION OF THE
TIMES-FOUR CENTRAL PROCESSOR

The times-four Central Processor (4XCP) is comprised of nine units -- one Instruction Processing Unit (IPU4) to process the CP commands, four Memory Buffer Units (MBU's) to provide central memory operands, and four Arithmetic Units (AU's) to perform the specified arithmetic operations. The structuring of these units is shown in Figure 1.

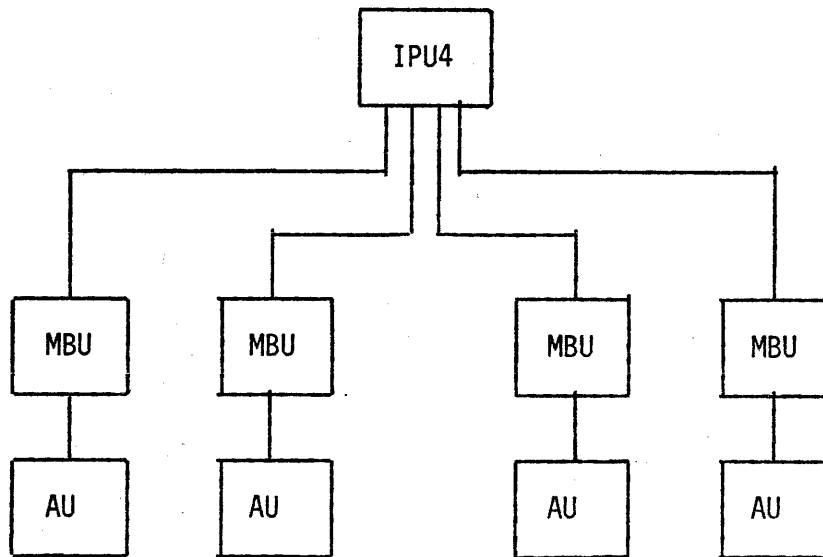


FIGURE 1. 4XCP UNIT STRUCTURE

The times-four IPU may also be used in a 1X, 2X, or 3X configuration to achieve the corresponding proportional increase in computational power. Figure 2 shows the unit arrangement for these three machine powers.

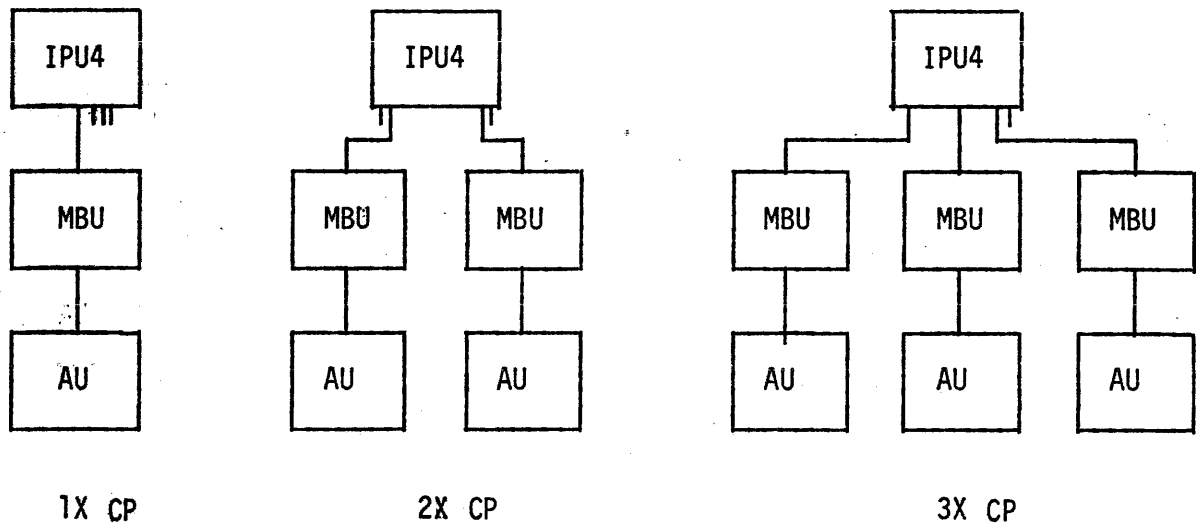


FIGURE 2. THE IPU4 CAN BE USED IN A 1X, 2X, OR 3X CONFIGURATION IN ADDITION TO THE 4XCP ARRANGEMENT OF FIGURE 1.

The 4XCP provides four parallel execution pipelines below the IPU. Any mixture of scalar or vector instructions may be in execution simultaneously in the four pipes. In any of the CP configurations, the interaction between an IPU, MBU, and AU is equivalent to that of one pipeline. The flow of data is from the IPU to the MBU, from the MBU to the AU, and then from the AU back to the MBU for stores to memory or back to the IPU for arithmetic results to the register file. The IPU is the only unit related to the configuration, and it performs all decisions pertaining to the routing of instructions to various pipes. MBU's and AU's are not aware of other MBU-AU pairs.

Four identical IPU-MBU interfaces exist at the IPU for data and control. A specific MBU is activated by control signals from the IPU when data is transferred to that MBU. Figure 3 shows the interaction of an IPU4, MBU, and AU for a 1XCP configuration.

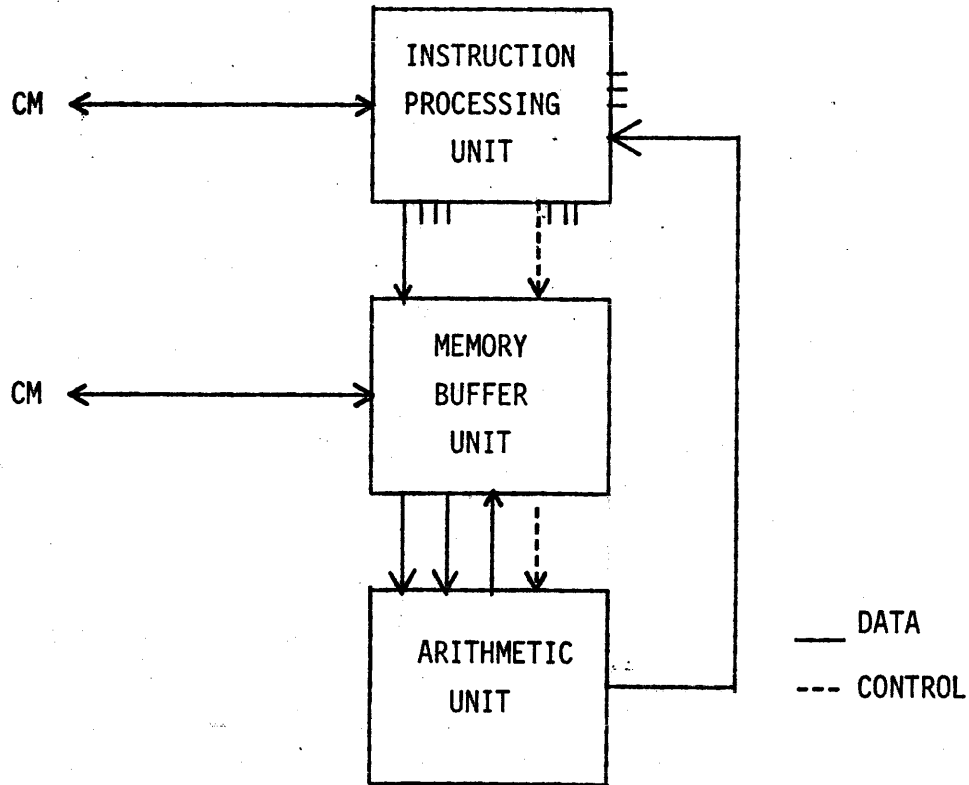


FIGURE 3. ARRANGEMENT OF UNITS FOR 1XCP.

In the multiple pipeline CP's, each MBU has its own dedicated memory port. The times-four CP, for example, uses five memory ports - one for the IPU4 and four for the MBU's.

The AU details information is loaded from or stored into memory only during maintenance commands and context switching. AU memory requests, therefore, occur infrequently and are routed through an expander cascaded on another expander as in Figure 4.

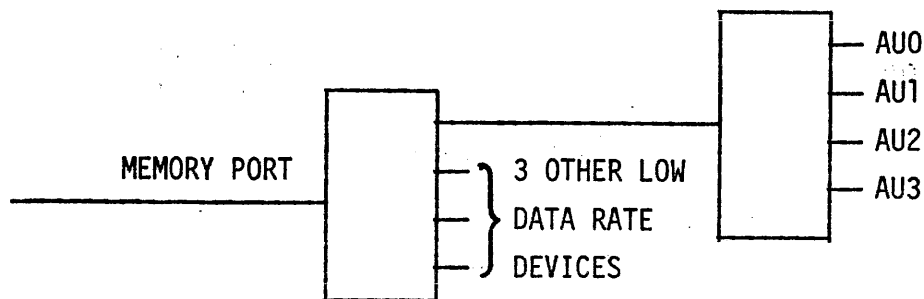


FIGURE 4. AU DETAILS MAP IS LOADED OR STORED INTO MEMORY THROUGH EXPANDERS.

The function of the IPU1, MBU, and AU is covered in Section B1 of the ASC Hardware System Manual. Section B1 of this manual gives a general description of the times-one Central Processor and will serve useful in providing a basic understanding of the CP in preparation for the material on the times-four block diagrams that follow.

IPU4 Sectional Description

The Instruction Processing Unit (IPU4) for the 4XCP contains six functional sections. These are shown in the block diagram of Figure 5 and listed following:

Instruction Address Development

Instruction and Register Files

Instruction Processing

Register Stack

Register Comparisons

Level 0 Through 4 Control

These functional sections are now described.

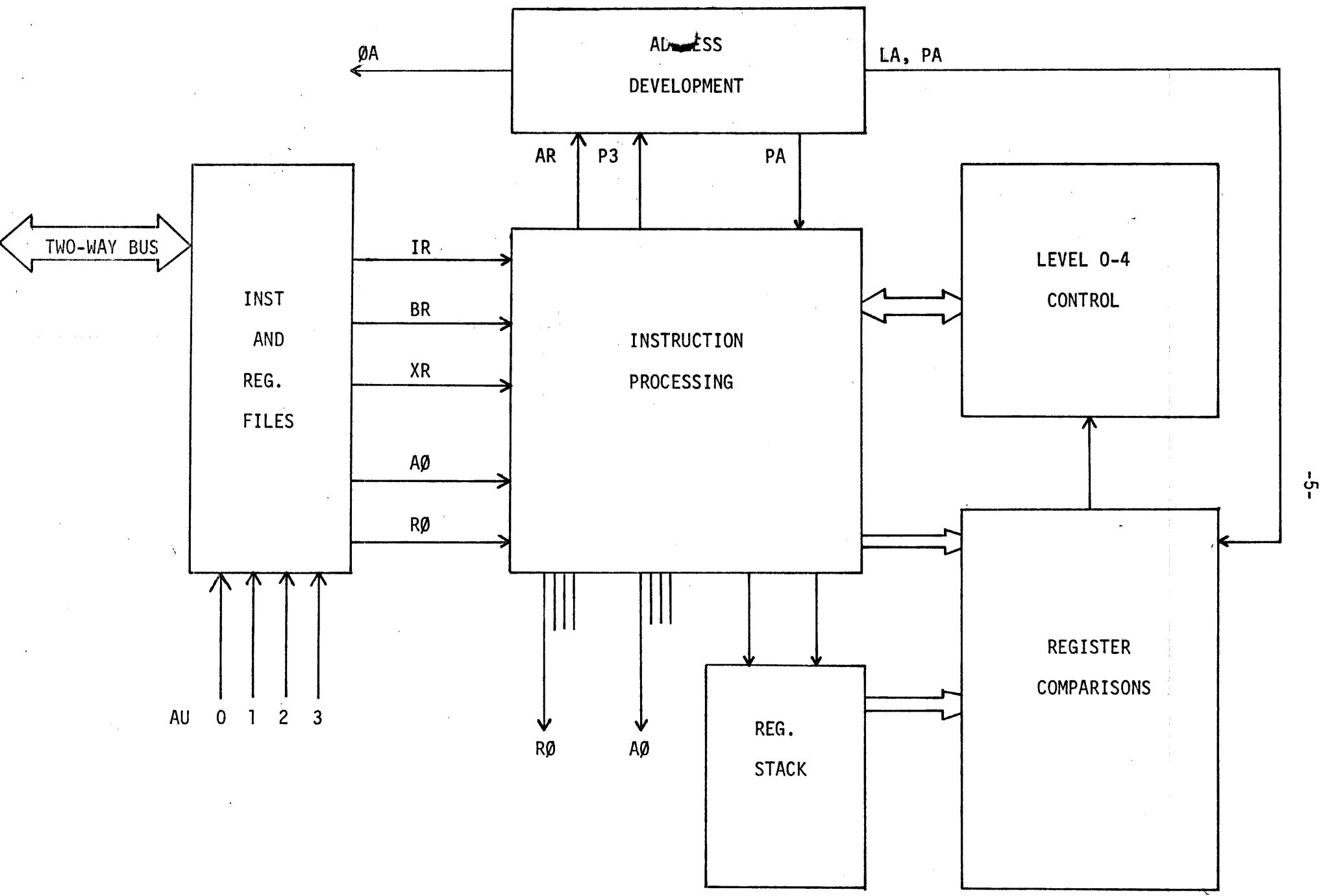


FIGURE 5. INSTRUCTION PROCESSING UNIT BLOCK DIAGRAM

- Instruction Address Development Section

The functions of the Instruction Address Development Section are as follows:

- Maintain the current instruction address
- Generate the look-ahead instruction address
- Save the LLA and PB instruction address
- Make all IPU memory requests

Figure 6 shows the registers and data paths of the instruction address development section. The present address register (PA) holds the instruction address of the instruction currently being fetched from one of the instruction files, KA or KB. PA is incremented by one each clock that a new instruction is loaded into level 1 of the instruction processing pipeline.

A branch instruction at level 3 loads its branch address into PA, LA, and ØA when a branch is taken. LA is the look-ahead register and normally keeps one octet ahead of the present address in PA. ØA holds the memory address while it is being sent to the MCU over the memory address lines. ØA is the register through which all IPU memory requests are made for instructions, indirect cells, objects of executes, load file, store file, and vector parameter loads.

BA is the branch address register used to save the LLA and PB instruction branch address. The address in BA is requested when the length counter for the LLA and PB instructions counts down to a point indicating that BA should be requested instead of the normal next look-ahead address in LA. After BA is requested, the next downstream instruction address at the target branch is saved. This provides a recovery address so that the

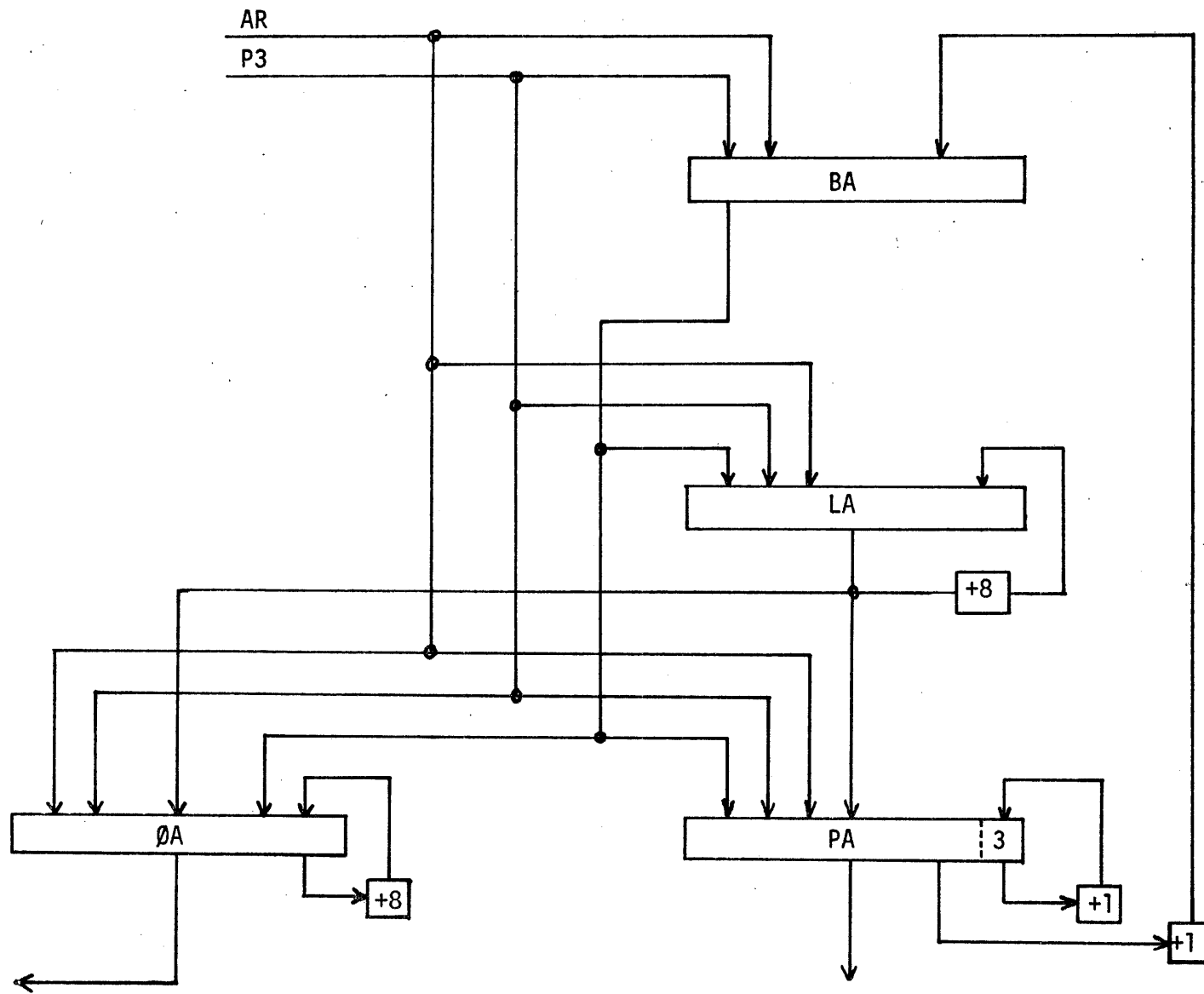


FIGURE 6. INSTRUCTION ADDRESS DEVELOPMENT SECTION

program can continue at an address one past the address of the target branch, should the branch fail to be taken.

- **Instruction and Register Files**

The instruction files are the KA and KB buffers of the IPU. Each buffer holds eight 32-bit instruction words. These buffers are used alternately such that a set of eight look-ahead instructions are being fetched from memory for one buffer while a set of eight current instructions are being read from the other buffer. The LA and PA addresses correspond to instruction octets associated with the KA and KB buffers. The sense of the relationship (LA with KA and PA with KB or vice versa) changes with each crossing of an octet boundary as instructions are read from the instruction files.

Figure 7 is a diagram of the instruction and register files. In Figure 7, KCM is a set of eight synchronizing registers to capture instructions or data on the two-way MCU data bus. KCM also holds the octet for an indirect address request or a request for the object of an execute instruction. When KCM is used for executes or indirects, the instructions in KA or KB are not disturbed. A path (line 1) is shown in Figure 7 from KCM to the instruction register (IR) at level 1 of the instruction processing section for routing indirect cells or execute objects to the instruction processing pipeline.

The register files are the program-addressable registers of the CP. These files are separated into octets (eight 32-bit registers) labeled

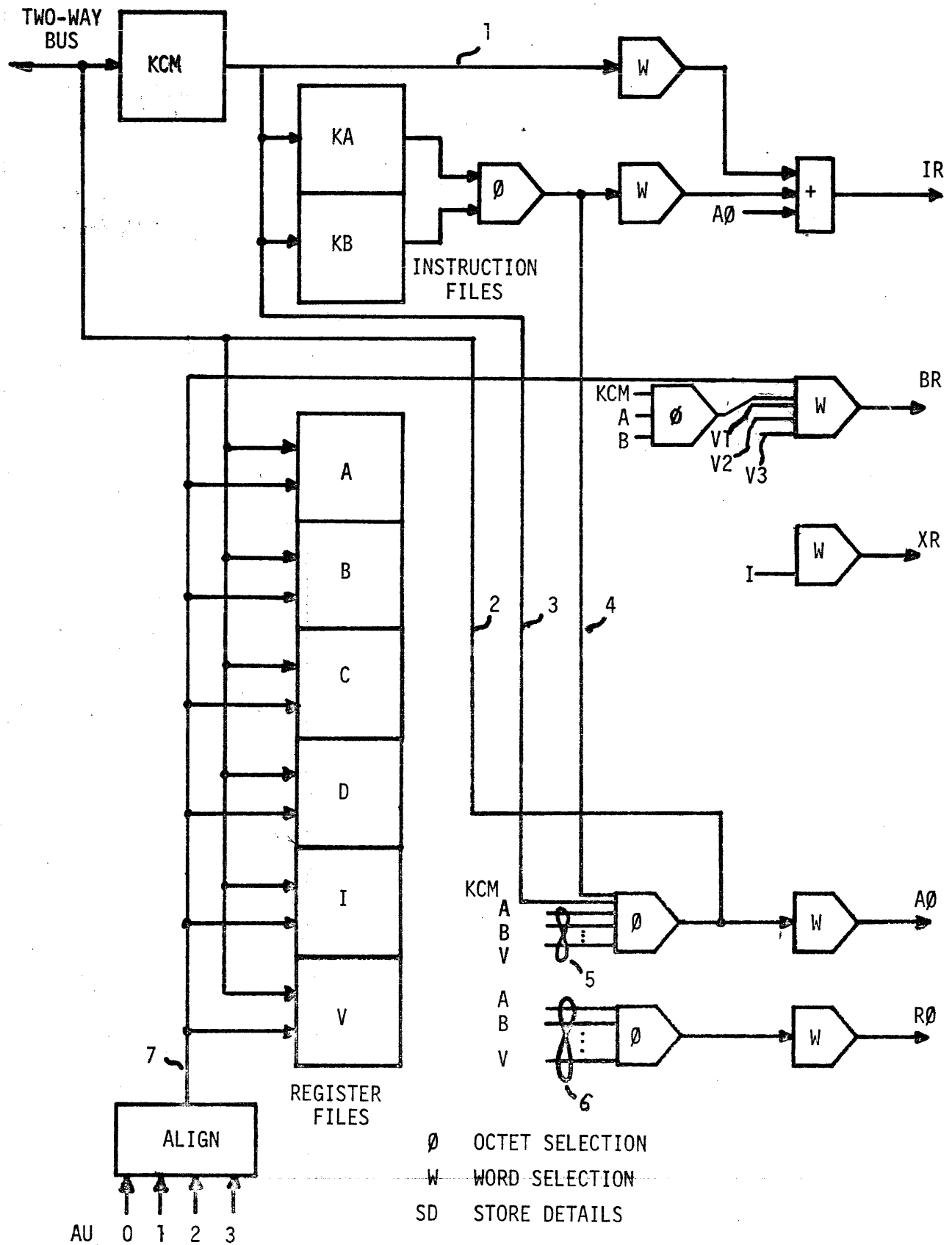


FIGURE 7. INSTRUCTION AND REGISTER FILES

A, B, C, D, I, and V. A and B are the set of fifteen base registers (base register 0 does not exist). C and D are the set of sixteen arithmetic registers. I is the set of eight index registers, of which only seven can be used in indexing operations. Index register 0 cannot be used for indexing but can be used for scratch pad operations. V is the set of eight vector parameter registers.

The output lines to the right side of Figure 7 are inputs to the following list of registers in the instruction processing section:

- IR Instruction register
- BR Base register
- XR Index register
- A \emptyset Address output
- R \emptyset Register output

Inputs to the register file may come from KCM in the case of a Load File instruction, from another file of the register file in the case of file-to-file operations, or from one of four Arithmetic Units in the case of AU results. Load File from memory operations (line 3) or file-to-file operations (line 5) use the octet selection network of A \emptyset and then feed data to the register file inputs over line 2. AU outputs are routed to the alignment hardware in the lower left of Figure 7 and then to the register file inputs over line 7.

- Instruction Processing Section

Four pipelined levels are contained in the IPU. The registers and data paths of the instruction processing pipeline are shown in Figure 8. The basic function of each of these four levels is as follows:

<u>Level</u>	<u>Function</u>
1	Instruction decode
2	Base and index selection
3	Address development
4	Register operand selection

Level 1 accepts one 32-bit instruction word at a time from the KA and KB instruction buffers. In parallel with this instruction transfer, the instruction address contained in the present address register, PA, of the instruction address development section is moved into register P1 of level 1.

Program counter values are held at each of the first three levels of the pipe and correspond to the address of instructions at each level. The instruction address is used at level 2 if the instruction is a program counter relative branch. The address is also used at level 3 if the instruction is a Branch and Load Base or Index type operation (BLB or BLX). These data paths are shown in Figure 8 from P2 to the ADDER in level 2 for the case of program counter relative operations and from P3 to the Address Output register (A0) of level 4 for the BLB and BLX operations.

Instruction decode is accomplished at level 1 by means of a read-only memory (ROM), shown in Figure 8 between the registers of level 1 and 2. Each instruction arriving at the IR register of level 1 applies its 8-bit operation code field to the address input of ROM1. These eight bits address one of 256 ROM locations. The ROM output is a 32-bit word containing control information for instruction processing at level 2. Another ROM is driven at level 2 by op code information handed down from level 1 when the instruction at level 1 moves to 2. These two ROM's provide operation decoding for various signals used to sequence controllers

at levels 2 and 3 of the instruction processing section. Control register C4 receives a small portion of its information from C3 and the remainder from control signal outputs of the level 3 controller. These signals from C4 control the interface to the Memory Buffer Units (MBU's).

Turning to the registers on the left side of Figure 8, we find the effective address development hardware. The effective address is formed from the addition of an index, base, and displacement value that is entered into the XR, BR, and NR registers, respectively, of level 2. These registers are entered with their selected values as the instruction which selects them moves from level 1 to level 2.

The selection of XR and BR is done by the selection networks in the Instruction and Register Files Section shown previously in Figure 7. The NR register receives a copy of the N-field of the instruction word; this was contained in register IR on the previous clock, assuming that instructions move through IR at the rate of one per clock. This assumption is made for ease of description and does not represent the real case for an ordinary mix of instructions.

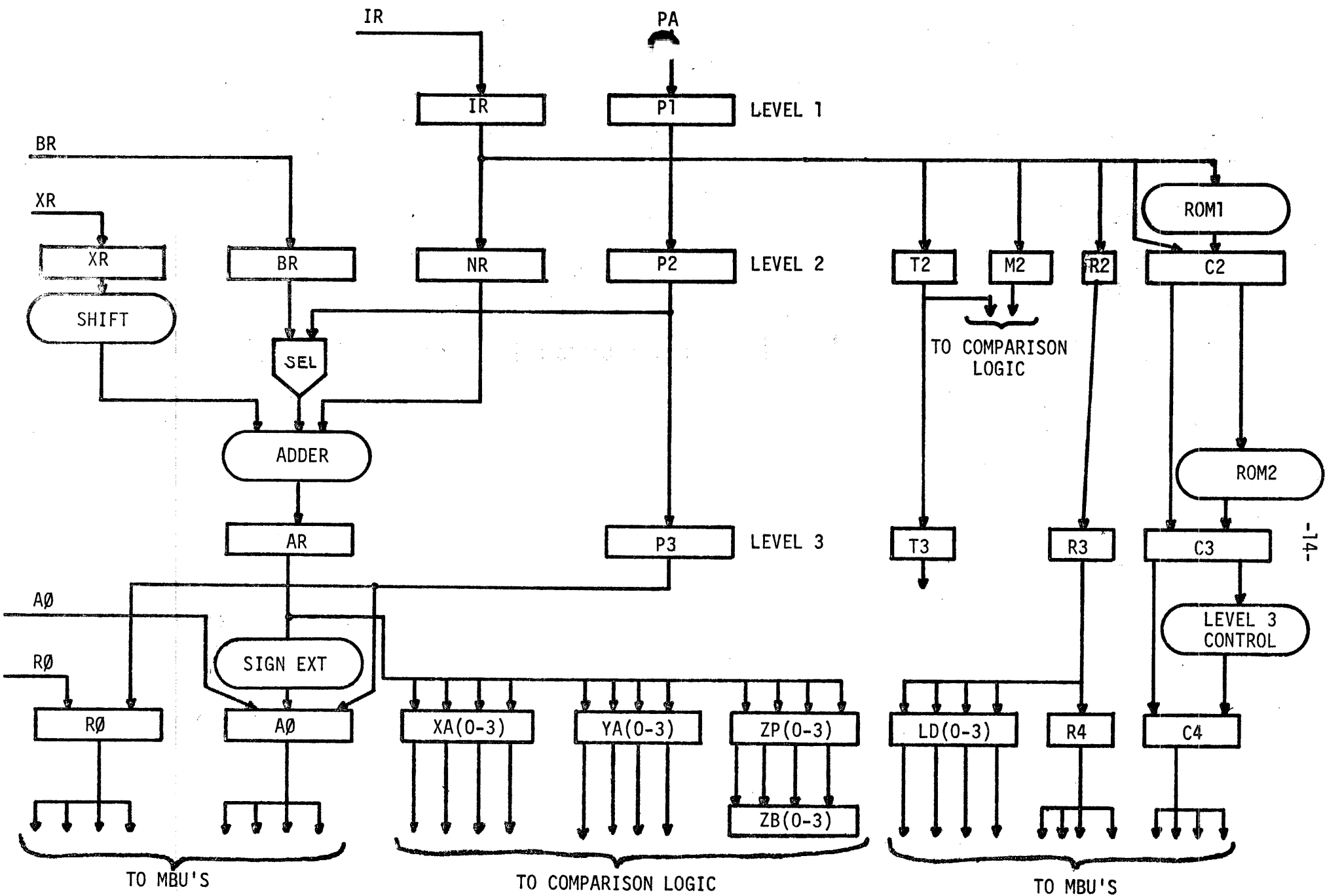
The shift logic following the XR register performs displacement indexing for different word size instructions. For example, a doubleword instruction causes a left shift of one bit position before the index register value in XR is added to the sum of BR and NR. A singleword operation causes no shift, and a halfword operation causes a one-bit right shift.

Addition of XR, BR, and NR takes place in the 3-input adder of level 2. The result of this addition is placed in the address register, AR, of level 3. For branches using program counter relative branch addresses, the program counter value in P2 becomes one of the inputs to the 3-input adder, replacing the base address in BR.

Immediate operand development also employs the adder of level 2 to perform the summation of an index plus an immediate value from the N-field of the instruction word. Arithmetic immediate operands require sign extension if the value is negative. This is carried out by the sign extension hardware at the output of register AR. From the immediate operand specifications for singleword operands, the sign extension occurs over the eight most significant bits of AR. Immediate operands are passed on to the MBU from the A \emptyset register at level 4.

Instructions that ordinarily reference memory may reference the register file if the effective address (alpha) is less than or equal to 2F (hex.), and the M-field of the instruction word is equal to zero. If $\alpha \leq 2F$ and $M = 0$, then a register of the register file is selected by the A \emptyset selection network of Figure 7 and entered into the A \emptyset register at level 4. This alpha register operand is sent to the MBU in parallel with the register operand from register R \emptyset . R \emptyset selection is also shown in Figure 7, the output of which appears as an input to the R \emptyset register of Figure 8. R \emptyset denotes register output, and A \emptyset denotes address output from the IPU to the MBU. A \emptyset is used for memory operand addresses to the MBU for ordinary memory referencing instructions when the condition, $\alpha \leq 2F$ and $M = 0$, is not satisfied.

Register AR has twelve additional outputs going to four XA registers, four YA registers, and four ZP registers. The XA and YA registers hold the memory octet addresses of resident data in the X and Y buffers of the four MBU's. Subsequent addresses arriving in AR are compared with the addresses of resident data contained in these X and Y buffers. Eight comparisons are made, four of the type AR versus XA(p) and four of the type AR versus YA(p), where p represents pipes 0 through 3. If a com-



-14-

FIGURE 8. INSTRUCTION PROCESSING SECTION

parison is found, then a memory reference is not made because the data is already resident in a read buffer of one of the four pipes. Pipe selection is partially based on the residency of read and write data within the MBU's.

The ZP(p) and ZB(p) registers represent the Z-pipe model and are used to keep track of memory write addresses for scalar data to be stored as the data passes down through the MBU and AU portions of the central processor. A Store instruction entering pipe p transfers its memory storage octet address from the effective address register (AR) of level 3 to the Z-pipe register, ZP(p), of level 4. This address is held in ZP(p), while the data to be stored passes through the CP pipeline into the Z buffer of the MBU. The octet address remains in ZP(p) as long as the data is resident in Z. As this data is forced out of Z into ZB due to a scalar write into another octet (entering pipe p) or due to a vector operation starting in pipe p, the address in ZP(p) is transferred to ZB(p). The octet address remains in ZB(p), while the memory write operation is taking place. Register ZB(p) becomes inactive again as soon as the write cycle has completed. The usage of these Z-pipe registers for monitoring read-write address conflicts is explained more fully in the hazard comparison description.

Continuing with a description of the registers shown in Figure 8, the T2 and T3 registers are seen linked to the instruction register, IR. The T-field of the instruction word in IR is copied into T2 and then into T3 as the instruction passes down through levels 2 and 3. At level 1 the three least significant bits of the T-field in IR are used to select an index register. The contents of the selected index register

are entered into the XR register of level 2. Bias bits are supplied by hard wired lines to the most significant end of register T2 in order to select the index file of the register file. Also, T2 is used to locate index register hazards for the case of a Store at level 3 writing into an index register.

T3 is used for index or arithmetic register selection by special register modification, test, and branch instructions (BCG and BCLE). For these instructions, T3 selects both a register to be used as an addend for the modification as well as a register to be used for the limit value in the comparison test. These two registers are selected from an even-odd register address pair, so the T-field of the instruction word must be even.

M2 is used to detect base register hazards by comparing the value in M2 (biased by an amount to select the base register file) with an address developed by a Store instruction at level 3. The Store instruction must have an effective address in AR that points to the needed base register (alpha $\leq 2F$ and $M = 0$) for the hazard to exist.

Registers R2, R3, and R4 carry the address of the register operand specified by the instruction word. The R-field in IR is four bits, specifying one of a set of sixteen possible registers. The register set is specified by the type of instruction; i.e., arithmetic, index, or base type. Two bias bits are appended to the most significant end of the R-field, while one bit is appended to the least significant end of the four-bit R-field. These three additional bits come from ROM1 and appear as decoding of the instruction type. The three bits are inserted at level 2 and included as part of registers R2, R3, and R4 forming a

seven-bit register address. The six MSB's specify one of 48 singleword registers of the register file, while the LSB is used to specify the left or right halfword for halfword instructions.

R3 contains the register address which is used for source register selection. This address is applied to the R \emptyset selection network of Figure 7. The output of the selection network goes to the R \emptyset register at level 4. Doublewords are selected into R \emptyset . Alignment of singlewords and halfwords from this doubleword is done by the Memory Buffer Unit.

Another output from R3 goes to the register hazard comparison logic described shortly under "Register Comparisons." R4 copies R3 and forms the beginning of the register stack which drives the register comparison logic.

Registers LD(0) through LD(3) are the four last destination registers. LD(p) is updated with the address of the register being modified. This information is needed for detecting a condition where the last register modification instruction to have entered a given pipe happens to be modifying the source register needed for the current instruction at level 3. Rather than waiting for the modification to take place in the register file, it is possible to use special hardware paths from the output of the Arithmetic Unit (AU) to the input of the AU. Using this path, the current instruction can pick up the modified register value at the input to the AU. Therefore, the current instruction may proceed to the AU input of the pipe containing the required register value and obtain that value via the AU short-circuit path without waiting at level 3.

This completes a description of the instruction processing section of Figure 8. We proceed now to the Register Stack shown in Figure 9.

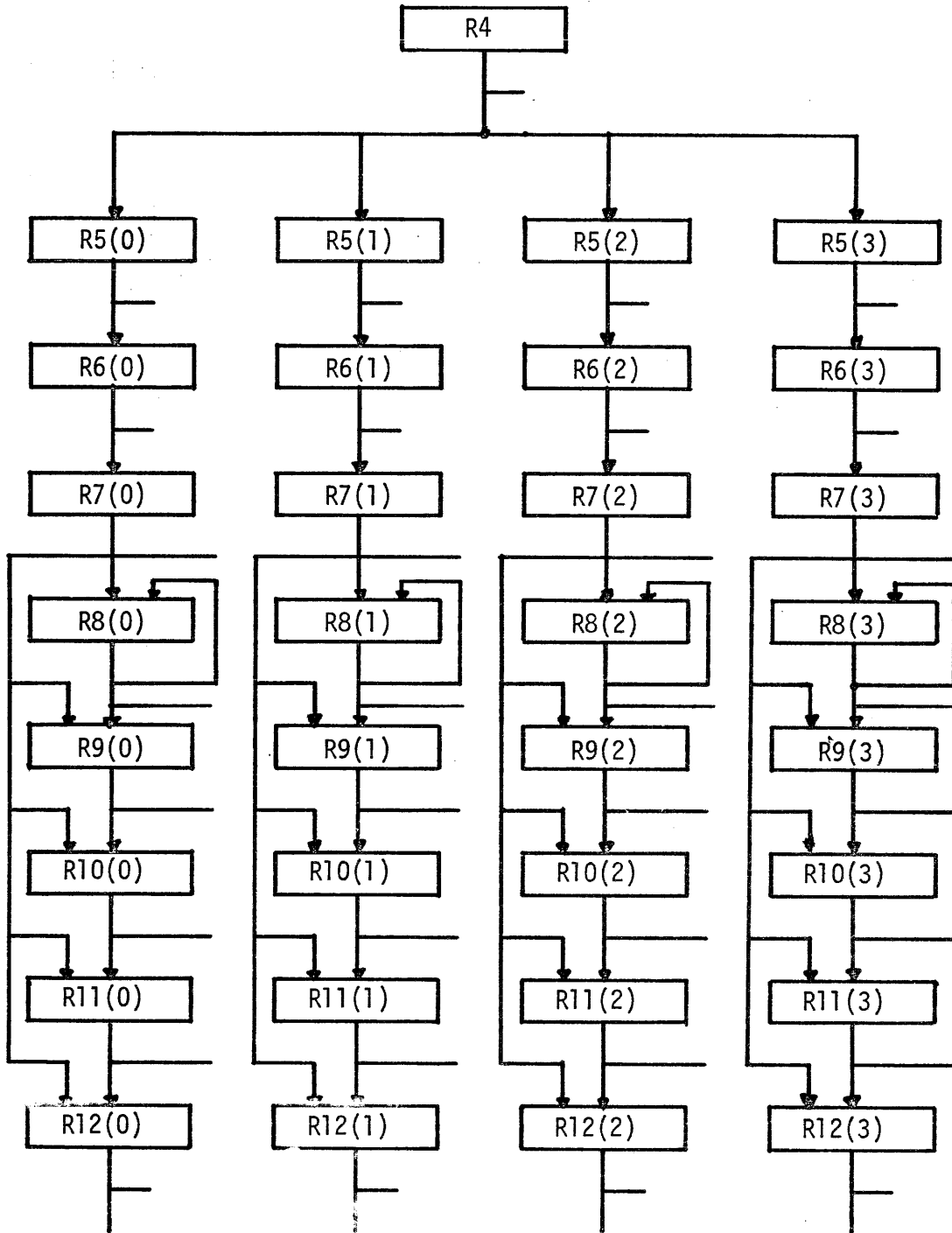


FIGURE 9. REGISTER STACK

Register Stack

Destination register addresses are kept in the register stack; this stack holds the addresses of registers to be modified by instructions still being processed below level 3 of the CP pipeline. The register stack begins with one register at level 4; it then becomes four registers wide at level 5 and continues this width through level 12. A width of 4 is provided to allow four pipes to process scalars in parallel. The singular register (R4) at the top of Figure 9 is the same R4 register as shown in Figure 8.

For a given pipe, each register of the register stack holds the register destination address of an instruction currently being processed at each level. These instructions have data which is destined to modify one of the program-addressable registers of the register file (files A, B, C, D, I, and V). Other information pertaining to a given instruction is kept in the register stack and travels along with the instruction as it proceeds down the pipe. This other information deals with the type of instruction; i.e., word size, result code setting, compare code setting, arithmetic exception possibilities, Z-store type instructions, etc.

The register stack has two main purposes: (1) it provides the necessary control signals at the AU output level for routing data either to the IPU or to the MBU; (2) it supplies the register comparison section with the register destination addresses of instructions below level 3.

Registers 5, 6, 7, and 12 of the register stack correspond to the MBU input, MBU output, AU receiver, and AU output levels, respectively. Registers 8, 9, 10, and 11 represent internal levels of the Arithmetic Unit but are not in a one-to-one relationship with sections of the AU. These registers are intended for timing; that is, to keep track of in-

structions while they are passing through the internal levels of the AU. Figure 9 shows the output from register 7 going to the input of registers 8, 9, 10, 11, and 12. These paths allow the timing from AU receiver (level 7) to AU output (level 12) to be varied from one to five clocks. The one-clock path goes directly from 7 to 12, while the five-clock path goes through all registers (from 7 through 8, 9, 10, 11, and 12). For example, a one-clock AU operation is a Load, Store, or Logical instruction which goes directly from the AU receiver to the AU output level. Logical operations are performed in the AU output section. A floating-point add instruction takes five clocks in the AU and uses all registers of the register stack. The AU sections used for the floating-point add are the Exponent Subtract, Align, Add, Normalize, and Output. Most scalar instructions fall within the AU timing range of one to five clocks. However, the floating-point doubleword multiply and all divide instructions take longer than five clocks in the AU. Tracking of these instructions is accomplished by holding data at level 8 of the register stack until four clocks before the end of the operation. In this manner the register address and control information arrive at level 12 of the register stack simultaneously with the arrival of the AU result at the AU output.

Outputs from each register of Figure 9 go to the destination inputs of the register hazard detection logic of Figure 10. The destination inputs enter from the left side of each of the four pipe columns shown in Figure 10. Source register inputs appear at the top of each column. Each of the source register inputs (T1, M1, R3, and AR) is repeated four times for each matrix of comparisons, whereas the destination inputs come from separate registers of the four register stacks. The symbol,

SOURCE →

DESTINATION ↓

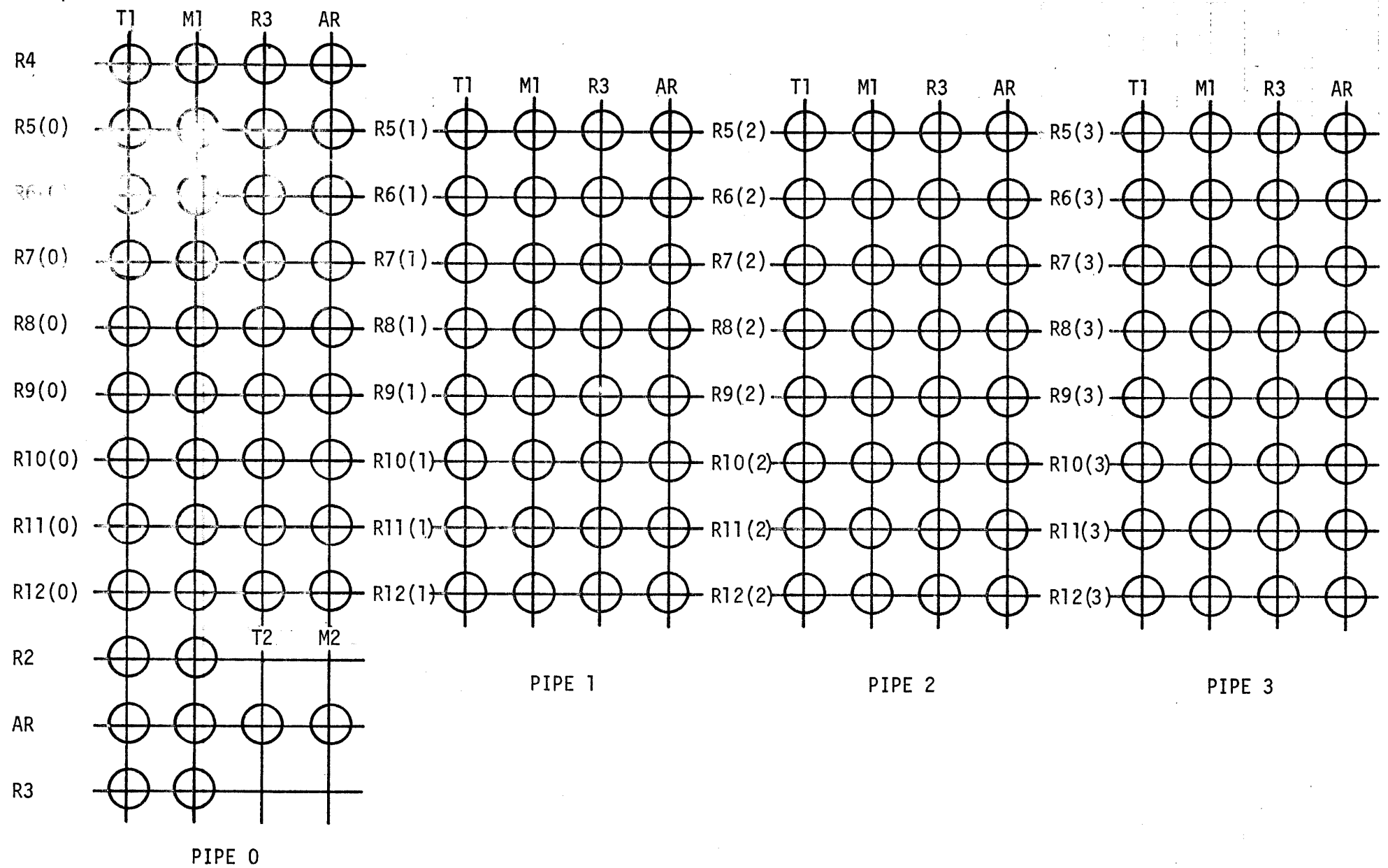


FIGURE 10. REGISTER HAZARD DETECTION LOGIC

T2 vs. AR	Near range index hazard
M2 vs. AR	Near range base hazard

Last destination comparisons are shown in Figure 11. In this figure the LD(p) registers hold the register address of the last instruction entering pipe p that had a register destination. Comparisons from LD are made against both R3 and AR. R3 contains the register operand source address, and AR contains the alpha register operand source address. Only the seven LSB's of AR become involved in the comparison and those only when the address in AR is less than or equal to 2F and the M-field of the instruction is zero ($\text{Alpha} \leq 2F$ and $M = 0$). It is this condition which specifies that the address in AR will be used to address a register of the register file.

A true comparison of LD(p) versus R3 indicates that it is possible for the instruction at level 3 to proceed down pipe p and pick up its register operand at the Arithmetic Unit rather than to wait for the last register destination instruction of that pipe to place its result in the register file. The process of picking up an operand at the AU involves using the AU "short-circuit" path from the output to the input of the AU. Two paths are provided, one for register short circuits (LD versus R3 comparisons) and the other for alpha register short circuits (LD versus AR comparisons). Outputs from these eight comparisons are used in the scalar routing logic that determines which pipe will be selected for a given instruction at level 3.

The logical structure of the comparison to determine X, Y, and Z buffer residency as well as instruction and operand hazard checking is shown in Figure 12. This figure is composed of registers LA and PA from

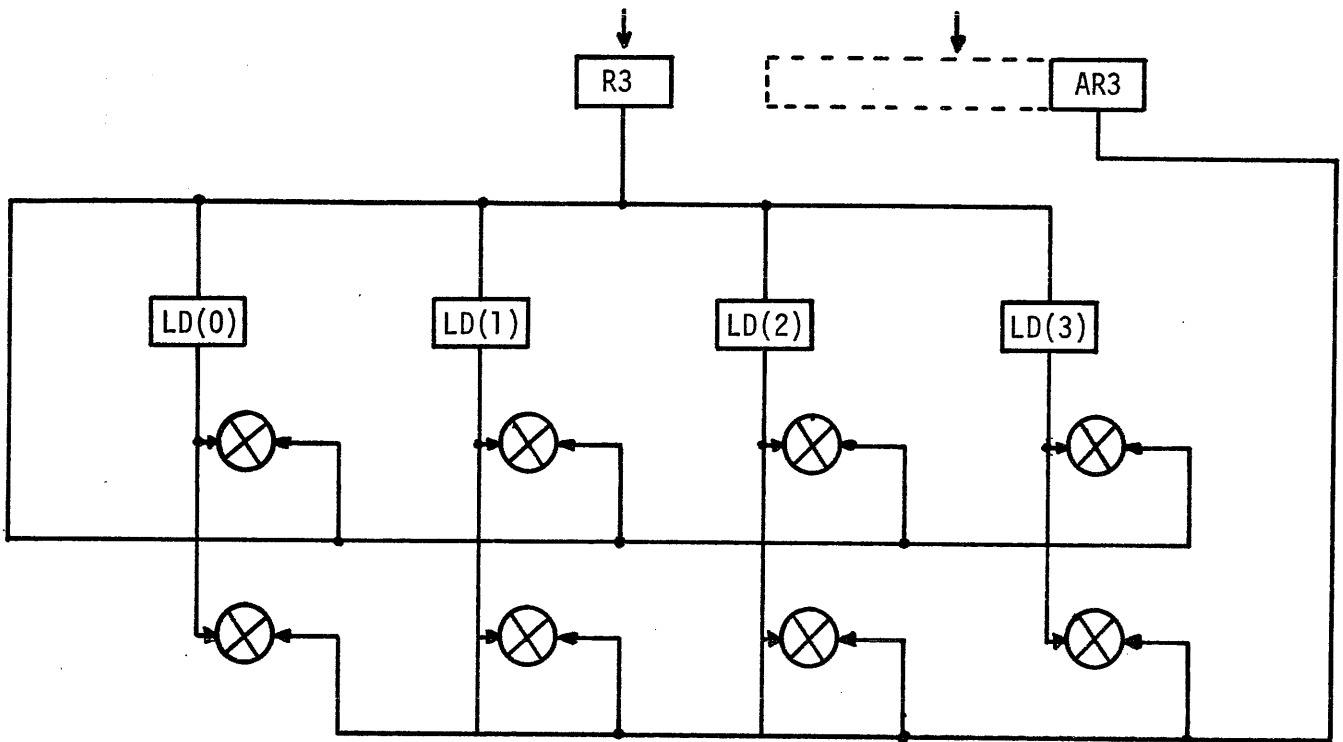


FIGURE 11. LAST DESTINATION COMPARISONS

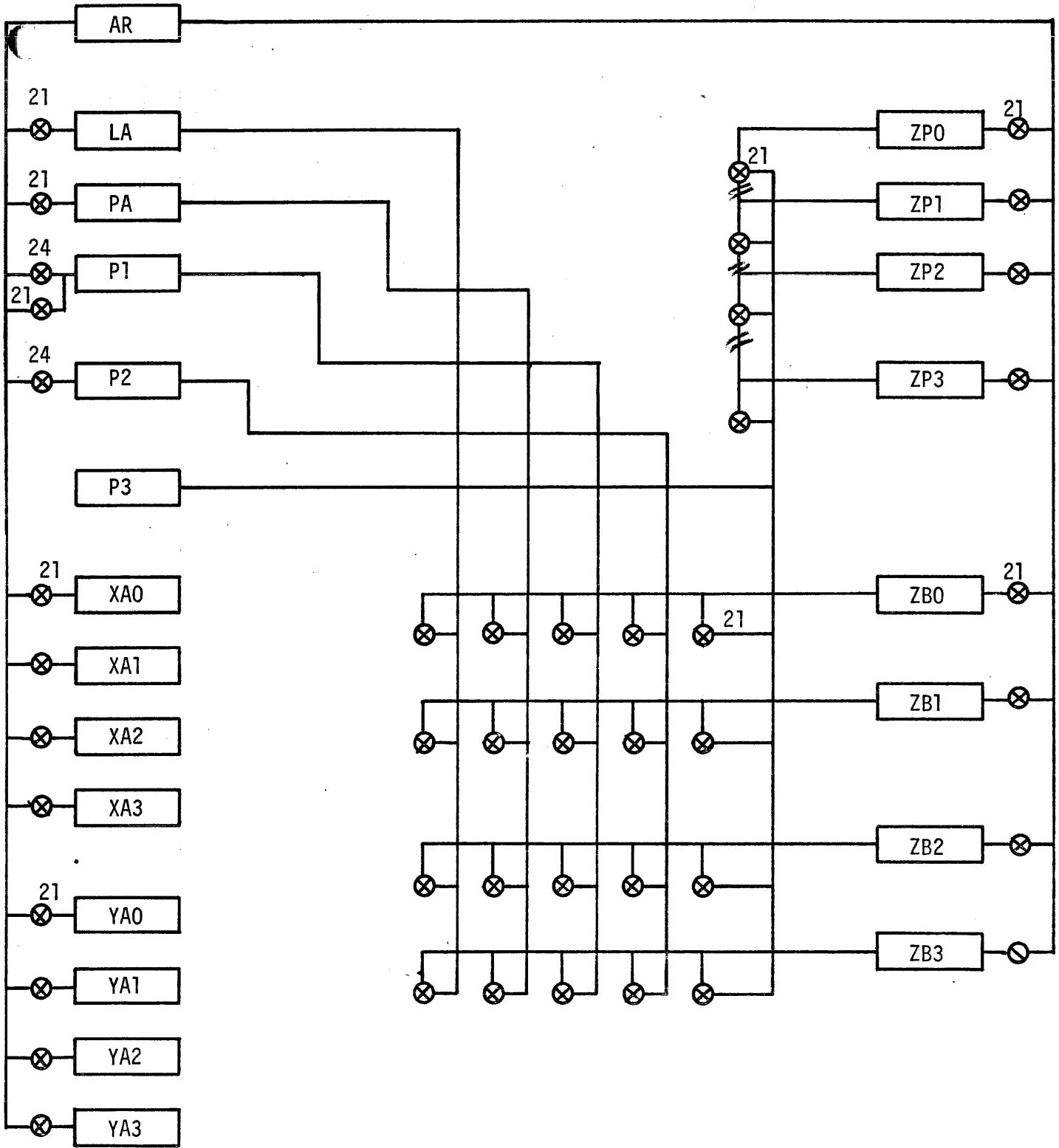


FIGURE 12. HAZARD COMPARISONS

Figure 6 and registers P1, P2, P3, AR, XA(0-3), YA(0-3), ZP(0-3), and ZB(0-3) from Figure 8. The first comparison on the left side of Figure 12 is LA versus AR. This comparison is for the purpose of detecting instruction hazards caused by a Store instruction at level 3 with its storage address in AR. Since LA contains the look-ahead address of instructions already requested from memory, a true comparison of LA versus AR means that at least one instruction in the octet addressed by LA will be an old instruction. It will not contain the modified instruction caused by the Store into that location. An octet comparison (21 bits) is made since the entire octet will have to be refetched regardless of the number of instructions modified within that octet. This involves a comparison of bit positions 8 through 28 of register AR, where the bit positions of AR are numbered from 0 through 31 from left to right.

PA versus AR works similar to LA. Both PA and LA mark their instruction hazards adjacent to the file that buffers the instructions (files KA and KB). When instructions are read from a file having its hazard flag set, this flag is picked up and travels down through levels 1, 2, and 3 with the instruction. Instruction hazard recovery is not initiated until the hazard flag reaches level 3, since a branch instruction ahead of the instruction with hazard could deflect the instruction path away from the hazard such that a refetch of the flagged octet is no longer required.

In addition to the AR versus LA and PA comparisons for the detection of hazards, these comparisons are used to find resident instructions in the KA or KB buffers for cases when a Branch instruction is at level 3 and the branch octet address in AR is equal to the address in LA or PA.

When this occurs, the branch address is not requested from memory. Instead, the address simply replaces the present address in PA; and the KR tag is toggled, if necessary, to the buffer containing the resident branch octet.

Program counter registers P1 and P2 are compared against AR across the full 24-bit address. This comparison detects conditions when a branch instruction at level 3 is branching to an instruction that is presently resident in levels 1 or 2. If AR equals P1, then the branch is either to program counter location ~~\$+2~~ and both levels 1 and 2 are active, or the branch is to ~~\$+1~~ and level 2 is not active. In the first case the instruction at level 2 is skipped and a branch is taken to the instruction at

four YA versus AR comparisons. These serve the function of locating resident X or Y octets. Registers XA 0 through 3 contain the octet address of data resident in the X buffer of pipes 0 through 3. From the instruction processing section of Figure 8, it was observed that the octet address of a read in register AR is passed to the XA(0-3) or YA(0-3) registers from AR when the read request was made. Active and full flags are kept on each of these octet address registers. Similar statements are true for registers ZP(0-3) when a Store instruction moves from level 3. That is, the first four comparisons in the upper right-hand column of Figure 12 (AR = ZP(0-3)) are for the purpose of locating resident Z storage octets. Data corresponding to these addresses are resident in the Z buffer of each Memory Buffer Unit. The next four comparisons (AR = ZB(0-3)) monitor the octet address of data in the process of being stored into memory from the Z buffers. This check is needed so that read requests in level 3 will be held up until data resides in memory for cases when the read octet is the same as the octet being written.

Other important facts are obtained from the set of sixteen comparisons consisting of AR versus XA, YA, ZP, and ZB. For example, when both AR versus XA(0) and AR versus ZP(0) comparisons are true, the implication is that both a read and a write to the same octet have occurred in pipe 0. Should another read request arrive for the resident X octet, no memory request will be issued; and pipe zero will be told to do a Z to X update. The Z to X update moves data from the Z buffer into X according to words of Z that have been modified. That is, if one word of Z has been written, then only one word of X needs to be updated.

In cases where a read address exists in AR and AR equals ZP(0) but AR does not equal any XA or YA, then the read request for the memory octet must be made and received prior to the occurrence of a Z to X update.

The four comparisons on the left side of the ZP registers of Figure 12 detect near-range instruction hazards. This hazard is termed "near range" because of the proximity of the instruction at level 3 to the Store instruction causing the hazard. Also included in the set of near-range hazards are the four comparisons connecting P3 to the ZB(0-3) registers. Any octet address agreement between an instruction address in P3 and a Storage address in ZP(0-3) or ZB(0-3) indicates that the instruction at level 3 is being modified by a prior Store operation.

If the comparison true is from ZB, then the store is currently in the process of being written to memory; and the instruction hazard recovery process can start as soon as ZB clears. ZB clears when data has reached its destination in the memory module. If the comparison true is from ZP, then the store octet is either still in the pipeline or is currently resident in the Z buffer of the MBU. Before the process of instruction hazard recovery can start, the pipe must first clear itself of stores, then initiate a forced write request to purge the Z buffer of its instruction modifying contents. It is observed that P3 is the "last change" point for the detection of instruction hazards. Past this point, instructions are committed for execution. For example, a Store instruction that modifies the very next instruction would be detected by means of this near-range hazard logic.

The remaining sixteen comparisons of Figure 12 are the far-range hazards. These sixteen comparisons consist of the matrix whose horizontal inputs are the four ZB(0-3) registers and whose vertical inputs are the four instruction address registers - LA, PA, P1, and P2. The term "far range" has reference to the space between instructions detected by this set of hazard logic, the furthest range being the comparison between the look-ahead register LA and the four ZB(0-3) registers. Each instruction

address register (PA, P1, and P2) at lower levels of the pipe become progressively "nearer" to the store that caused the instruction hazard.

The action taken for the process of instruction hazard recovery is similar to that just described for near-range hazards, the only difference being the fact that the process of instruction hazard recovery does not start until the instruction experiencing the hazard reaches level 3. That is, a hazard detection by LA or PA is simply marked by a flag associated with the KA or KB buffer holding the group of instructions, one or more of which has just been modified. A flag is used for P1 and P2 comparisons at levels 1 and 2. These flags are carried with the instruction as it moves down the instruction processing pipe to level 3. If the instruction with a hazard flag reaches level 3, then the instruction hazard is called and the process of recovery starts. This is done so that extraneous memory requests will not be issued for cases where a branch is taken prior to the arrival of hazardous instructions at level 3.


This completes a description of the hazard comparison logic of Figure 12. It also completes a description of all sections of the instruction processing unit block diagram of Figure 5, except for the level 0-4 control section. This section is described by a set of detailed flow charts for each level (0 through 4) of the IPU.

MEMORANDUM

10 April 1973

TO System Planning Group
Compiler Group

COPY TO Dennis Best Al Riccomi
Gary Boswell Charles Stephenson
Gary Cobb Dave Stevens
Bill Cohagan Marvin Talbott
Richard Hatcher Hollie Thompson
Mike Miller Freddie Walker
Sid Nolte

FROM  Bill Kastner

SUBJECT TIMES FOUR CENTRAL PROCESSOR HAZARD CONDITIONS

A description of the Times Four Central Processor hazards is included as an attachment to this memo.

This document lists the types of hazard conditions that occur in the times-four CP. These hazards are divided into three main categories: (1) those involving only scalar instructions, (2) those in which both scalars and vectors are in progress at the same time, and (3) those involving only vectors. These categories are further subdivided in the outline on the following page.

This information is intended primarily for the Compiler Group, but it also has more general application as a description of Central Processor behavior.

Bill Kastner

BILL KASTNER

BK:jc

Attachment

I. Scalar Hazards

A. Register Hazards

1. Register Operand Hazards
2. Alpha Register Operand Hazards
3. Index or Base Hazards
- 4.* Destination Hazards
- 5.* Largest Word Size Hazard

B. Central Memory Address Hazards

1. Store-Read Hazards
2. Store-Load File Hazards
3. Store-Store File Hazards
4. Store-File and Load-File R-Octet Hazards
5. Store File Hazard
6. Store File-Read Hazard

C. Instruction Hazards

1. Storing Over Instructions
2. Store File Over Instructions
3. Vectors Storing Over Instructions

D. Arithmetic Exception Hazards

1. Load Arithmetic Exception Mask or Condition Hazards
2. Store Program Status Hazards

E. Branch Hazards

1. Result Code Hazard
2. Condition Code Hazard
3. Arithmetic Exception Branch Hazard

*New 2X, 3X, and 4X Hazards

II. Scalar-Vector Hazards

- A. VPF Modification
- B. Scalars Writing Over Vector Input Arrays
- C. Vectors Writing Over Scalar Read Data
- D. Alpha Hazards During Vectors in Fork and Join Mode

III. Vector Hazards

- A. Vectors Storing Over Their Own Input Data Arrays
- B. Addressing Conflicts Between Two Vectors Executing Simultaneously in Parallel Pipes
- C. Halfword Z-fill-in Hazard

Throughout this description, operand and instruction conflicts are referred to as hazards. The intended meaning of "hazard" as used here should be "something to avoid if possible." In most cases not staying away from hazards will result only in a slower execution rate due to instruction delays while waiting for the hazard to clear, but the hazard will not affect the logical results of a program. In other cases, such as those involving the "vector hazard rule," a hazard is quite critical and will result in numerical program errors or loss of program control if the hazard rule is not taken into account. The difference in terminology between these two uses of the word "hazard" is distinguishable in this description by a block to the right of each hazard heading which indicates the following:

DELAY

for delay-producing hazards which do not affect program results.

FUNCTIONAL

for error-producing hazards.

I. Scalar Hazards

A. Register Hazards

1. Register Operand Hazards

DELAY

Two conditions are necessary for a register operand hazard to exist. These conditions are:

- (a) A first instruction of an instruction stream having a register destination aimed at one of the registers of the register file. This instruction is located below level 3 of the CP pipeline but has not yet entered its result into the register file.
- (b) A second instruction, occurring at a later point in the instruction stream, having arrived at level 3 of the CP pipeline, and having a register source requirement to read a portion or all of the register presently destined to be modified by the first instruction.

Hardware solution: Logic is provided to detect this hazard. Register operand hazards are cleared when the first instruction has modified its destination register.

Delay avoidance: It is possible to avoid some of the delay due to register operand hazards. Two methods exist.

- (a) Using the short-circuit path -

Two short-circuit data paths exist from the output to the input of the arithmetic unit. One is for Register Short Circuits, and the other is for Alpha Register Short Circuits.

The Register Short-Circuit path is used when a second instruction experiencing a register hazard determines that a first instruction causing the hazard is still the last instruction to have entered a given pipe. Other instructions may have occurred between the "first" and "second" instructions, but none of these have taken the same pipe as the first instruction.

For this short circuit to take place, the word sizes of the source and destination registers of the two instructions must be the same. An exception to this rule is that halfword-to-halfword short circuits are not provided. In addition to the same word size short circuits, there also exists short circuits for doubleword results feeding back to second instruction singleword register sources with even register addresses.

(b) Instruction insertion -

In some cases it may be possible to insert other instructions between the two that cause a register operand hazard. The hazard is not actually avoided by this method; it is just postponed to the point where it does not exist any more. The method does, however, allow the CP to perform other useful computations during time that the CP would normally be waiting for the hazard to clear. Of course, the other work performed could not use the register which is causing the hazard.

Delay time: Register operand hazard delay is dependent upon the pipe time of instruction I1 which is 4 clocks + AU time + memory time, providing the AU output does not become blocked due to multiple AU outputs destined for the register file. AU time is found listed in Table 1 of the CP timing section (B2) of the CP Hardware Volume. Memory time is eight clocks if instruction I1 makes a memory read request. This time delay equation assumes an initially empty pipe. For accurate timing estimates, all pipe conditions prior to instruction I1 must be taken into account.

Example of register operand hazard:

<u>Inst #</u>	<u>Inst R, α</u>	<u>Pipe</u>	<u>Comment</u>
I1	LOAD A1, CM1	(0)	
I2	LOAD A2, CM2	(0)	Assume AR=ZP(0)
I3	ADD A1, CM3		

For this hazard to appear in the 4X CP, operation of the AU short-circuit path must be blocked by one or more other instructions using a different register between the two instructions that cause the hazard. The other instruction(s) must use the same pipe in order to block the short-circuit path. For purposes of this example, it is assumed that a prior store instruction has left the Z-buffer of pipe 0 with data destined for octet CM2 in central memory. This condition forces instruction I2 down pipe 0, thereby blocking I3 from picking up its register operand (A1) over the AU register short-circuit path from the prior result

of instruction I1. Without the short circuit, I3 must wait in level 3 until A1 has been modified by instruction I1.

2. Alpha Register Operand Hazards

DELAY

Two conditions are necessary for an alpha register operand hazard to exist. These conditions are:

- (a) A first instruction of an instruction stream having a register destination aimed at one of the registers of the register file. This instruction is located in the CP pipeline between levels 4 through 12 inclusive.
- (b) A second instruction, occurring at a later point in the instruction stream, having arrived at level 3 of the CP pipeline and having an effective address $\alpha < 2F$ and an M-field of zero, such that an alpha register source requirement exists to read a portion or all of the register presently destined to be modified by the first instruction.

Hardware solution: Logic is provided to detect this hazard. Alpha register operand hazards are cleared when the first instruction has modified its destination register.

Delay avoidance: The same two methods that were used to avoid delays due to register operand hazards are also useful for avoiding delays due to alpha register operand hazards.

Delay time: Alpha register hazard delay is dependent upon the pipe time of instruction I1 which is 4 clocks + AU time + memory time, providing the AU output does not become blocked due to multiple AU outputs destined for the

register file. AU time is from Table 1, and memory time is eight clocks if instruction I1 makes a memory read request.

<u>Inst #</u>	<u>Inst R, α</u>	<u>Pipe</u>	<u>Comment</u>
I1	LOAD A1, CM1	(0)	
I2	LOAD A2, CM2	(0)	Assume AR=ZP(0)
I3	ADD A3, A1	(0)	

In this example instruction I2 takes pipe 0 because its central memory read data, CM2, is assumed to be resident in the Z-buffer of pipe 0. I2 blocks the possibility of instruction I3 picking up its alpha register operand (A1) via the AU alpha register short-circuit path from the output of instruction I1. Without the short circuit, I3 must wait in level 3 until A1 has been modified by I1.

3. Index or Base Hazards

DELAY

Two conditions necessary for an index or base hazard are:

- (a) A first instruction of an instruction stream having a register destination aimed at one of the index or base registers of the register file. This instruction is located in the CP pipeline between levels 2 through 12 inclusive.
- (b) A second instruction, occurring at a later point in the instruction stream, having the requirement to use the index or base register presently destined to be modified by the first instruction.

Hardware solution: Logic is provided to detect this hazard. Index or base hazards are cleared when the first instruction has modified its destination register. Index or base hazards hold the second instruction at level 1 until the hazard is cleared. However, late index or base hazards occur when a Store instruction into a base or index register immediately precedes a second instruction requiring the index or base register at level 2. For late hazards of this type, the second instruction will refetch its index or base register at level 2 when the hazard has cleared.

Delay avoidance: There is no hardware short circuit to decrease the delay due to index or base hazards. The method of instruction insertion can be used to perform other work while waiting for the index or base hazard to clear.

Delay time: Index or base hazard delay is dependent upon the pipe time of instruction I1 which is 6 clocks + AU time + memory time, providing the AU output does not become blocked due to multiple AU outputs destined for the register file. AU time is from Table 1, and memory time is eight clocks if instruction I1 makes a memory read request.

Example of Index hazard:

<u>Instruction #</u>	<u>Inst R, α</u>
I1	LOAD X1, CM1
I2	ADD A1, CM2, X1

In this example, instruction I1 modified index register X1; then the next instruction, I2, uses index register X1 to develop its effective address, $CM2 + (X1)$. The second instruction must wait in level 1 until instruction I1 has modified index register X1.

4. Destination Hazards

DELAY

Destination hazards exist in a 2X, 3X, or 4X CP, but not in a 1X CP. This hazard is due to a Load, Interpret, or Normalize instruction having the same register destination address as that of some prior instruction which has not yet entered its result into the register file. The prior instruction is any type that has a register destination.

If this hazard were not detected by the 2X, 3X, and 4X machines, then it would be possible for a second instruction to overtake a first instruction, via another MBU-AU pair, and place its result in the register file prior to the result of the first instruction. Results occurring out of sequence in this manner would not leave the latest value in the register file at the common register address of the two instructions. Hardware protects against this type of hazard.

Hardware solution: Logic is enabled in a 2X, 3X, or 4X configuration for detecting destination hazards. This logic is disabled in a 1X configuration, since one instruction cannot pass another instruction in a one-pipe machine. Also, hardware is provided in the 2X, 3X, and 4X machines to force Load type instructions down the pipe in which the destination hazard exists for cases when the effective address is selecting a register of the register file ($\alpha \leq 2F$ and $M = 0$). Forcing the instruction into the pipe

behind the destination hazard has the effect of preventing the instruction race condition, while at the same time allowing the Load type instruction to proceed without experiencing a destination hazard delay.

For the case when the effective address of a Load type instruction is selecting a central memory operand, a destination hazard causes further checking by hardware to determine whether the Load address (in register AR of level 3) is equal to the Z-octet address contained in pipes other than the one causing the destination hazard. If there is address agreement, then the destination hazard causes a delay until the hazard clears. This delay prevents a forced write of Z in the other pipe and a read request for the Load address from the destination hazard pipe. If there is no address agreement, then the Load takes the pipe causing the destination hazard, and no delay occurs.

Delay avoidance: Delays due to destination hazards are avoided by hardware for the cases when: the Load address is selecting a register of the register file; or when the Load address is selecting central memory and the address is also resident in the Z-buffer of the destination pipe; or when the Load address is not resident in the Z-buffer of any of the pipes.

The method of instruction insertion can be used to fill in the dead time waiting for a destination hazard to clear.

Delay time: A destination hazard clears when the register destination instruction causing the hazard has placed its result in the register file. For meaningful code, the register destination instruction (I1) should be followed by a Store, then by the Load type instruction (I3) experiencing the delay. Otherwise, I3 will simply write over data entered into the destination register by I1. If, for example, I1 were an ADD, then the result of the addition would be lost because of the Load into the same register.

With the Store instruction between I1 and I3, the destination hazard delay due to I1 is 3 clocks + AU time + memory time, providing the AU output does not become blocked due to multiple AU outputs destined for the register file.

Example of destination hazard is as follows:

<u>Instruction #</u>	<u>Inst R, α</u>
I1	DIVIDE A1, CM1
I2	LOAD A1, CM2

In this example, the divide instruction could be replaced by any instruction with a register destination of A1. The Load does not see a source register operand hazard because its source operand is from central memory. It would be all right for the Load to take a different pipe if it were not possible for the Load to overtake and pass the divide. To guard against this possibility, the Load instruction is routed down the same pipe taken by the divide. The divide pipe is taken assuming that CM2 is not

resident in the Z-buffer of any other pipe. If residency exists, then the Load waits at level 3 until the destination hazard due to the divide has cleared.

Note that this example shows meaningless code, since the result of the division is destroyed by the Load.

5. Largest Word Size Hazard

DELAY

Four conditions must exist for a largest word size hazard to occur. These conditions are:

- (a) Only two types of second instructions can cause this hazard. They are Multiply instructions (Op codes 6C and 7C) with an even addressed arithmetic register specification and fixed to floating point conversion instructions (Op codes AA, A9, and AB; mnemonic codes FXFD, FHFL, and FHFD, respectively). These two types of instructions are the only ones which use a larger register destination word size than their own source word size.
- (b) The second instruction must be preceded by a first instruction with a register destination of smaller word size than that of the second instruction's register destination.
- (c) The register destination of the first instruction must be contained within the register modification space of the destination register of the second instruction.
- (d) The register destination of the first instruction does not have the same address as the source register of the second instruction. Otherwise, if the addresses were the same, then the hazard would be classified as a register operand hazard.

Hardware solution: Hardware detects largest word size hazards and holds the second instruction in level 3 until the hazard clears.

Delay avoidance: The method of instruction insertion can be used to fill in the dead time waiting for a largest word size hazard to clear.

Delay time: Largest word size hazard delay time is dependent upon the time for instruction I1 to clear the pipe. This time is 4 clocks + AU time + memory time, providing the AU output does not become blocked due to multiple AU outputs destined for the register file. AU time is from Table 1, memory time is eight clocks if instruction I1 makes a memory read request.

Example of largest word size hazard:

An example of an instruction sequence with a largest word size hazard is given following in which a Load to an arithmetic register is followed by a multiply using an adjacent, evenly addressed arithmetic register.

<u>Instruction #</u>	<u>Inst R, α</u>
I1	LOAD A1, CM1
I2	MULTIPLY A0, CM2

The source register of the multiply does not show up as a register operand hazard with the destination register of the Load instruction. The multiply instruction, using even address register A0, will place its result into the even-odd register address pair A0-A1. If, for example,

the read octet of the multiply was resident in one of the four pipes and the read octet of the Load was not resident, then it would be possible for the multiply to place its doubleword result into registers A0 and A1 prior to A1 being entered by the Load instruction. If A1 were entered late by the Load, then the expected final doubleword product of the multiply would be overwritten in its least significant half. Largest word size hazard logic in the CP does not allow this to happen.

Another example is presented to show that the hardware does not call out a largest word size hazard unnecessarily. In this example, an evenly addressed register (A0) is used by the Load, while an odd register (A1) is used by the multiply. This is just opposite to the register usage of the previous example.

<u>Instruction #</u>	<u>Inst R, a</u>
I1	LOAD A0, CM1
I2	MULTIPLY A1, CM2

The multiply instruction does not have a source register operand hazard or a largest word size hazard. The multiply selects its pipe according to the scalar routing rules. There is no hazard delay for this instruction sequence.

B. Central Memory Address Hazards

1. Store-Read Hazards

DELAY

A Store instruction followed by a read from the same octet of memory causes store-read hazards. Several store-read sequences are considered in order to see how timing between instructions influences whether this type of hazard appears or not.

Example B11.1

<u>Instruction</u>	<u>Inst R, α</u>	<u>Pipe</u>
I1	STORE A1, CM1	(0)
I2	ADD A2, CM1	(X) $\begin{cases} X=0 \text{ if no SC} \\ X \neq 0 \text{ if SC} \end{cases}$

For this example, instruction I2 at level 3 determines that there is address agreement between address register AR and the Z-pipe register, ZP(p), for one of the pipes (p). The ZP registers cover Store instructions from levels 4 through 12 and the Z-buffer. Another set of four registers, ZB(p), holds the Store address of data being written into memory.

At this point in the determination of pipe selection, it is possible for instruction I2 to have a register operand hazard and find that it can pick up its register operand (A2) by using the AU short-circuit path. If an AU short circuit is found, then instruction I2 will initially try to take a pipe other than 0 because pipe 0 contains a last destination register address of A1. The register source

address of an instruction is placed in the last destination register for Store instructions if they do not modify data when passing through the arithmetic unit. Store Negative is an example of a Store type instruction that modifies its register source data in the AU.

If the Add initially tries to take a different pipe because of both a register hazard and a short-circuit condition, then a forced write to central memory will be started for octet CMI in pipe 0. This forced write occurs so that the data from location CMI can be read by another pipe. The only data path from the output of one pipe to the input of another pipe is by means of central memory. Before the read request can be issued by another pipe, the complete write cycle for that memory location must have finished. This is because the four pipes are connected to memory over separate memory ports, and the Memory Control Unit (MCU) does not guarantee that a write to memory will be processed first for the case of a write then a read to the same octet on two different ports.

The timing delay for both a write and a read cycle is shown as case 1 in the store-read timing diagram of Figure 1. The ADD is completed on clock 30 for case 1.

It is highly likely that the register will clear before the forced write is complete; in which case, the Add instruction may choose any other pipe for its read from location CMI. This change in routing does not affect the delay

time if the forced write has been initiated. However, if the register hazard clears before the forced write is initiated, then the read from location CM1 can be done from pipe 0. If octet CM1 is not resident in the X- or Y-buffers of pipe 0, then a memory read request is required before the Z-to-X update can occur. This is shown as case 2 of Figure 1, in which the Add is completed on clock 17. This is thirteen clocks earlier than the time for completion in case 1.

Case 3 occurs when the store immediately precedes the add, there is no short-circuit condition to cause the Add to select another pipe; and the X- or Y-buffer has a resident octet containing location CM1. The resident X or Y is the result of a prior memory read request for data within the octet containing word CM1. The Add is allowed to move into level 6 when there are no Z-stores in the pipe and the update from Z to X has taken place. The Add is completed on clock 12, some eighteen clocks earlier than case 1.

Case 4 is one in which there are no Z-stores between levels 4 through 12 of pipe 0, but pipe 0 contains a resident Z-octet of address CM1. This implies that the Store into CM1 occurred prior to the arrival of the Add at level 3 and that there were other instructions between the Store and the Add which allowed time for the Store to pass through the pipeline into the Z-buffer. Also, there is assumed to

be a resident X as a result of a prior memory read request. The Add is completed on clock 9 for this case. This is twenty-one clocks earlier than case 1.

The Store-read example used for the purpose of this description is functionally the same as

I1	STORE	A1, CM1
I2	ADD	A2, A1

Using this code, the Add takes the same pipe as the Store because of the alpha short circuit on register A1. There is no long write then read delay as previously described. Therefore, it is apparent from what has gone before that code should be written as in the preceding whenever possible in place of the Store-add example given earlier.

- CASE 1. STORE INTO CM1, THEN ADD WITH SHORT CIRCUIT TO ANOTHER PIPE, NONRESIDENT X.
- CASE 2. STORE INTO CM1, THEN ADD WITH UPDATE FROM Z TO X, NONRESIDENT X.
- CASE 3. STORE INTO CM1, THEN ADD WITH UPDATE FROM Z TO X, RESIDENT X.
- CASE 4. RESIDENT Z OF OCTET CM1, THEN ADD WITH UPDATE FROM Z TO X, RESIDENT X.

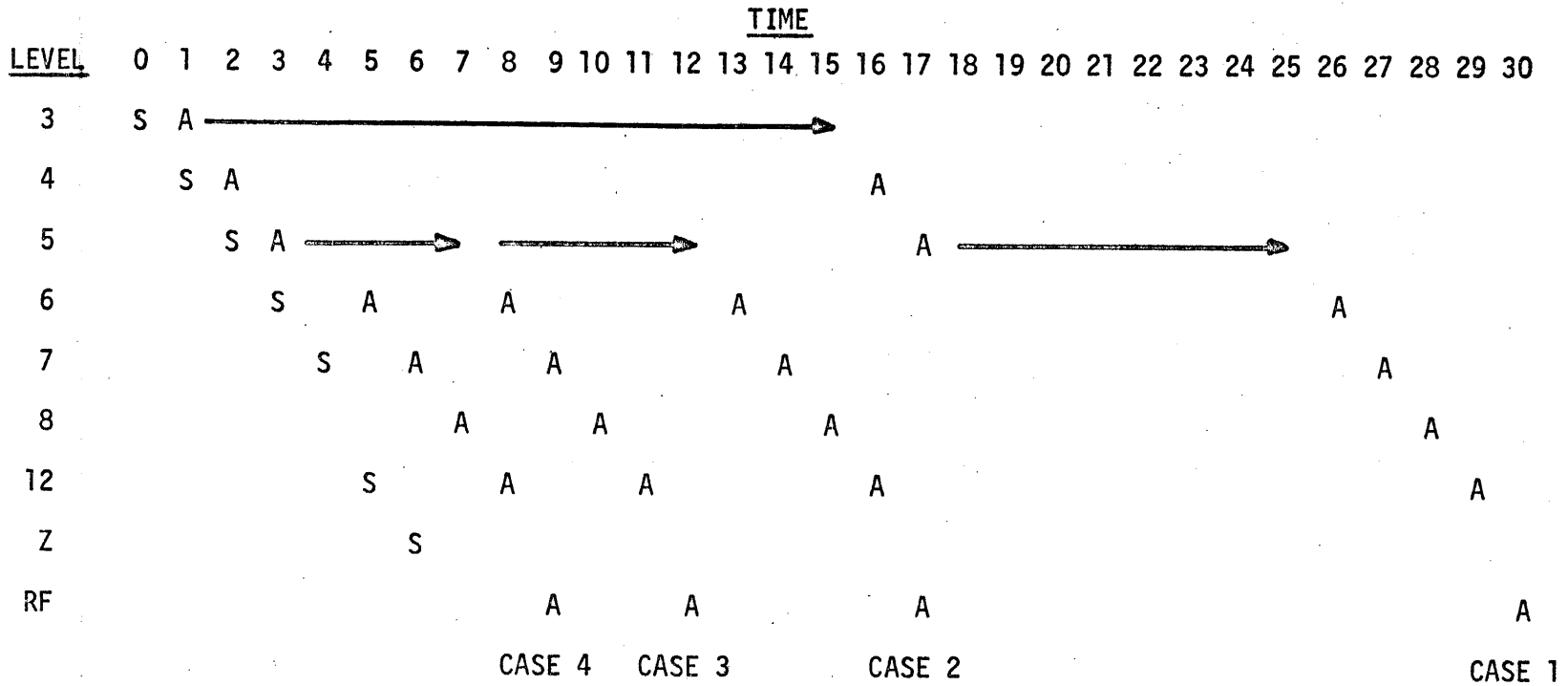


FIGURE 1. STORE-READ TIMING

2. Store-Load File Hazards

DELAY

This hazard is caused by CP code of the form :

Follows:

<u>Instruction #</u>	<u>Inst R, α</u>
I1	STORE A1, CM1
I2	LOAD FILE R, CM1

Store-Load File hazards are the result of a Load File instruction requesting the same octet from memory as was written into by a previous Store instruction. The Store could have been the immediately preceding ^{inst.} or one which occurred some time ago but which left its Z-octet resident in the pipe that was taken; i.e., no other Stores to different octets have taken that pipe since the Store to CM1.

Since the Store octet is resident in one of the pipes, its address, CM1, is in some ZP (p) register. The Load-File address, CM1, is compared against all ZP addresses when the Load File reaches level 3 of the IPU. Address agreement from this comparison is called an alpha hazard.

At this point the LF instruction at level 3 finds the pipe (p) in which the resident Z-octet of address CM1 is located. A check is made to see if the hazard is due to a vector in progress rather than a scalar Store to memory. If the hazard is due to a vector in progress, then the alpha comparison is ignored; and the Load File proceeds to execution. The alpha comparison is ignored in this case because compiled code will not contain vectors

that write over scalar data areas when the CP is placed in the FORK mode. If the CP is placed in the JOIN mode, then it is not possible for subsequent scalar instructions to proceed to execution until the vector has completed.

If a vector is not in progress in the pipe (p) in which the alpha hazard exists, then the LF instruction in level 3 waits until there are no Z-stores in pipe p. This wait insures that the Z-octet buffer has all its data before a "start forced write" command is issued to the MBU.

The LF instruction continues to wait at level 3 until the storage of octet CMI is completed to central memory. It then proceeds to execution at level 3, making its load file request via the IPU4 memory port.

Hardware solution: Logic is provided to detect this hazard. Store-Load File hazards are cleared when the Z-octet containing the Load-File data has been written into memory. The Load File is held in level 3 until the hazard is cleared; then it is executed at level 3. A Load-File instruction does not use the CP pipeline below level 3.

Delay avoidance: Delay cannot be avoided for adjacent Store-Load File instructions. The method of instruction insertion can be used to make the instructions nonadjacent. If this method is used, then a second Store to a different octet should be forced down the pipe containing CMI by means of a short circuit to the register address of the Store. However, be careful selecting a different octet for storage

because, if the different octet happens to be resident in some other pipe, then the Store will take that other resident pipe. Also, the X- and Y-buffers must not contain the address of the second Store (ST2) because then ST2 would have to wait on outstanding reads before the forced write of ST1 could be issued. Also, the second Store must not occur before the first Store has had time to pass through the pipe; otherwise, the second Store would have to wait in level 3; and that is what is to be avoided. The list of contingencies on the second Store would seem to preclude its use as an effective means to avoid a Store-Load File hazard; however, this method is given in hopes that it can be used to advantage.

Delay time: Store-Load File delay depends on the time for the first Store to pass through the pipe into Z, then for the Z-octet to be written into memory. This time is 5 clocks + memory write time if the Store immediately precedes the Load File. The five clocks are the pipe time of the Store and can be deleted if it is known that there are no Stores in the pipe; i.e., that the resident Z-octet has all its data. Also, this timing equation assumes that the AU output does not become blocked due to multiple AU outputs destined for the register file. This would be due to instructions prior to the Store. Memory write time in this equation is normally equal to eight clocks.

3. Store-Store File Hazards

DELAY

This hazard is essentially the same as the Store-Load File hazard just described. In fact, the same descriptions can be used for the hardware solution, delay avoidance, and delay time sections by replacing the words "Load File" with "Store File."

In order for this hazard to exist, the Store instruction's address must be contained within the octet address space of the Store File instruction. This is an unlikely condition for reasonable code, since data stored is immediately written over by the Store File instruction.

Prior to execution of the Store File, the Store instruction is forced out of the CP pipeline into memory so that the Store File is certain to place its data into memory after the store. This is done to preserve the order of instructions.

4. Store File and Load File R-Octet Hazards

DELAY

This hazard is common to both Load File and Store File instructions. The delay occurs when the octet selected by the Load File or Store File instruction at level 3 (as specified by the R-field) covers a register being modified by an instruction below level 3.

In the case of a Load File, the purpose of the delay is to insure that all registers have been modified for all instructions below level 3 that have register destinations targeted for the same octet as that specified by the Load File. Otherwise, if the delay were not applied, a late arriving register destination could modify a register within the octet entered by the Load File after the Load File was completed. The order of instructions would not be preserved if the Load File were executed without testing for an R-octet hazard.

For a Store File instruction, the purpose of the delay is to insure that the register file octet to be stored has been modified by all instructions below level 3 that have register destinations targeted for the same octet as that to be stored. This is so the Store File will have all its data to be stored.

5. Store File Hazard

DELAY

A Store File hazard is due to a memory read instruction followed shortly by a Store File to the same memory octet. The memory read is positioned in time such that it has requested memory but has not yet received its data from memory. With this condition true, the Store File cannot proceed to execution because, if it did, then it is possible for the Store File data to be written into memory prior to data being read from memory by the instruction using a memory operand. Since the read and write operations take place on different memory ports, the Memory Control Unit (MCU) cannot guarantee that a read arriving first will be processed first. For this reason, the Store File must wait until the memory data has been received.

Hardware solution: Hardware detects this hazard. Store File hazards are cleared when the X-or Y-buffer of the MBU, which had an outstanding request for an octet of memory of the same address as the Store File, has received its data. The Store File waits at level 3 until the hazard has cleared.

Delay avoidance: This delay can be avoided by separating memory reads to a given octet from Store Files to that same octet. The separation can be done by instruction insertion. Enough time must be allowed for the read data to be received before the Store File arrives at level 3 for execution.

Delay time: Store-File hazard delay depends on the memory time of the instruction causing the hazard. If the memory read instruction is immediately ahead of the Store File, then the Store-File wait time can be as high as ten clocks, assuming no memory interference.

Example: An example of CP code producing this hazard is as follows:

<u>Instruction #</u>	<u>Inst R, α</u>
I1	ADD A1, CM1
I2	STORE FILE R, CM1

6. Store File-Read Hazards

DELAY

This hazard is caused by CP code of the form:

<u>Instruction #</u>	<u>Inst R, α</u>
I1	STORE FILE R, CM1
I2	ADD A1, CM1

which is just the reverse of the code given in the previous example for a Store File hazard.

To be sure of getting the latest data, the Add must wait for the Store File to complete its write to memory before the Add can request memory. This hazard delay occurs as a normal part of a Store File instruction. That is, the second instruction is delayed regardless of its instruction type or memory address.

Hardware solution: This hazard is protected by the level 1 controller of the IPU. When a Store File instruction is detected at level 2, the level 1 controller goes into the "Store File state" and blocks any subsequent instructions from reaching level 2. Subsequent instructions are held at level 1 until the "write acknowledge" signal is received from the MCU and the "data available" signal has returned to "zero." These conditions indicate that the Store File data has been received by the MCU and that the process of writing data into the memory module has been initiated. At this point the instruction following the Store File is released from level 1.

Delay avoidance: This delay cannot be avoided; it is inherent in the operation of the Store File instruction.

Delay time: Store File read hazards cause a delay of eight clocks in the execution of the "read" instruction at level 3, assuming that there is no wait for "write acknowledge" or for "data available" due to memory interference. These eight clocks are to be considered as the time to execute the Store File and not as any additional delay caused by the fictitious Store File read hazard.

C. Instruction Hazards

1. Storing Over Instructions

DELAY

Scalar Store instructions that modify other instructions may cause an instruction hazard delay. In particular, when the address being stored into by a Store instruction (below level 3) is equal to the instruction address of an instruction at levels 1, 2, or 3 or is contained within the octet of instructions requested or resident in the KA or KB buffers, then an instruction hazard flag will be set.

Store instructions below level 3 include Stores resident in the Z-buffers of each of the four pipes. These Stores have not yet been written into memory. Stores in the pipeline or in the Z-buffers hold their storage addresses in the ZP registers. Stores being transferred to memory from the Z-buffers through the ZB-buffers hold their storage addresses in the ZB-registers during the transfer. The ZP- and ZB-registers are compared with the program counter values from level 3 upwards through the present address (PA) and look-ahead address (LA) registers. A true comparison indicates an instruction hazard.

Instruction hazard recovery takes place if the instruction so marked as having an instruction hazard reaches level 3 for execution. Level 3 is the point at which a flagged instruction hazard becomes a real in-

instruction hazard. This distinction is made because it is possible for an instruction to be marked as having a potential instruction hazard but to never require the instruction hazard recovery process as a result of a Branch instruction taking the branch prior to the flagged instruction's arrival at level 3. In this case the flagged instruction is not executed; and, so, there is no need to recover the modified instruction.

The process of instruction hazard recovery involves performing a forced write operation on the pipe that contains the Store which caused the instruction hazard. Once the Store octet has been written into memory, the IPU may refetch the instruction address that was marked as having an instruction hazard.

Hardware solution: Instruction hazards are divided into two classes for the purposes of hardware implementation. These classes are: near-range hazards and far-range hazards. A near-range hazard occurs when there is a true comparison between the program counter at level 3 (P3) and the ZP- or ZB-registers of any of the four pipes. This condition causes a forced write and then an immediate instruction hazard recovery. It is not a potential but a real instruction hazard when it occurs at level 3.

A far-range hazard is when there is a true comparison between the following pairs of registers:

Look-ahead	LA vs. ZP for all pipes
Present address	PA vs. ZP "
Level 1	P1 vs. ZP "
Level 2	P2 vs. ZP "

These comparisons are flagged as potential hazards at the appropriate level of the pipeline. A far-range hazard may never reach level 3 if a branch is taken before the flagged instruction arrives at level 3. However, if it arrives at level 3, then a forced write is sent to the pipe containing the Store that caused the hazard. Instruction hazard recovery starts after the forced write is complete. Recovery is accomplished by fetching the modified instruction from memory.

Delay avoidance: The simplest and most effective way to avoid instruction hazards is to abide by the rule "never modify instructions."

Delay time: For the case of a Store Negative (STN) instruction directly ahead of an instruction which the STN modifies, it takes six clocks for the Store to reach the Z-buffer from level 4, another six clocks to complete the forced write to memory, eight clocks to fetch the modified instruction into the KA- or KB-buffer, and three more clocks to get it down to level 3 where it was when the instruction hazard was detected. This totals twenty-three clocks for a worst case instruction hazard recovery.

2. Store File Over Instructions

DELAY

This hazard is basically the same as storing over instructions (CJ). The main difference is that the comparisons with program counter addresses P1, PA, and LA are made against AR instead of ZP and ZB. The octet address being stored into is in the AR register at the time the Store File is executed. This address does not pass through the ZP- and ZB-registers as does the address of a Store instruction. Also, there is no need for AR to be compared against the program counter registers at levels 2 and 3 (P2 and P3) because level 2 is blocked from holding any instructions during a Store File and level 3 is where the Store File is being executed; i.e., an instruction cannot modify itself.

No forced write is necessary as a part of Store File instruction hazards. Completion of execution of the Store File implies that the Store File octet is in memory. A refetch of the instruction address occurs when the instruction marked with an instruction hazard reaches level 3.

Hardware solution: Store File instruction hazards are detected by the hardware. AR is compared with P1, PA, and LA on an octet level. If the hazard reaches level 3, instruction hazard recovery starts immediately by fetching the modified instruction octet from memory.

Delay avoidance: Do not modify instructions.

Delay time: Since the instruction hazard recovery process does not start until the flagged instruction reaches level 3, the delay is equivalent to a Branch to a nonresident octet at that point in the program. This delay is eleven clocks.

3. Vectors Storing Over Instructions

FUNCTIONAL

Several possibilities exist for this hazard; these may be divided into two classes: (1) a vector storing over subsequent vector instructions and (2) a vector storing over subsequent scalar instructions. For both classes, the FORK/JOIN mode determines the way in which hardware deals with the hazard.

Consider a first vector instruction which has its "Allow Following" bit set to "zero" so that subsequent scalar or vector instructions are not allowed to start execution until all vectors in progress have completed. If the next (second) instruction is a vector, then it is allowed to request the vector parameter file and initialize the Memory Buffer Unit (MBU) of the selected pipe. Or, if the second instruction is a Load File (LF), then the operation of loading the register file can be completed; but that is all; no other instructions can be executed in the JOIN mode.

In either case, the hardware monitors for instruction hazards. This is done by comparing the \vec{C} vector addresses of all vectors in progress with the program counter of the instruction at level 3. If a true comparison is found, then the level 3 far-range hazard flag is set. This flag causes an instruction refetch of the modified second instruction after all vectors have completed.

If the second instruction is changed to something other than the instruction it was, then there is no way the register file can be reinstated to what it was before execution of the second instruction (the VECTL or LF executed while the first vector or vectors were running). If the conditions of this hazard have occurred as stated, then the program will lose control, or produce incorrect answers, as a result of the register file being modified by an instruction that was modified.

If the second instruction is anything other than a Vector or Load File, the hardware monitors for an instruction hazard. This is done by comparing the \vec{C} vector addresses of all vectors in progress with the program counter of the second instruction at level 3. A true comparison will set the level 3 far-range hazard flag, which - in turn - will cause an instruction hazard recovery request after all vectors in progress have completed. In this case no damage has been done since the second instruction at level 3 was never executed before its modification (as was the case with a VECTL or LF instruction). The modified instruction will be executed after the instruction recovery process has been completed.

Consider now a first vector instruction which has its "Allow Following" bit set to "one" so that subsequent scalar or vector instructions are allowed to proceed in parallel with the first vector. If the \vec{C} vector addresses

of the first vector happen to write over instruction addresses of subsequent instructions in or above level 3 (but within the IPU), then the near-range hazard signal or the far-range hazard flag will become true, resulting in a refetch of the modified instruction when the hazardous instruction reaches level 3.

This hazard has the potential of causing intermittent problems during checkout because a hazard may show up or not depending upon the phasing of C vectors with respect to instructions in the upper half of the pipe. However, to isolate the problem to some extent, the "Allow Following" bit can be set to zero and the level 3 far-range hazard flag checked at the completion of the vector to see if the next instruction address at level 3 has been overwritten.

Hardware solution: Instruction hazards are marked in the JOIN mode when vectors write over subsequent instructions in the IPU. These hazards may or may not occur, depending on vector-scalar phasing in the FORK mode.

Marked instructions make an instruction hazard recovery request to obtain the modified instruction.

Delay avoidance: Do not allow vectors to write over instruction areas of memory.

Delay time: Instruction hazard recovery time is eleven clocks. Recovery occurs after all vectors have completed for the case in which the last vector had the "Allow Following" bit set to zero.

If the original and modified instruction was a VECTL or LF, then the file load time is lost. Also, the second vector initialization time is lost if a VECTL or VECT is modified.

D. Arithmetic Exception Hazards

1. Load Arithmetic Exception Mask or AE Condition Hazards

DELAY

Three instructions exist which load the arithmetic exception condition or mask registers. These are:

LAM Load arithmetic exception mask

LAC Load arithmetic exception condition

LEM Load arithmetic exception mask and condition

These three instructions must wait until none or only one pipe contains instructions that may modify the arithmetic exception (AE) condition or mask registers. Instructions that modify the AE condition register are an LAC, LAM, or any arithmetic operation that has the potential of setting any of the AE condition bits. These bits are:

Divide check

Fixed-point overflow

Floating-point exponent overflow

Floating-point exponent underflow

Instructions that modify the AE mask register are an LAM or LEM.

If only one pipe contains instructions that may modify the AE condition or mask registers (these instructions generate an AE hazard signal in the pipeline below level 3), then that pipe will be selected, providing the effective address is big (to central memory, $\alpha > 2F$). If the address is small ($\alpha \leq 2F$ and $M = 0$), then the addressed register

must not currently be undergoing modification by another pipe. If no modification is taking place, then the pipe containing the AE hazard will be selected. If modification is taking place, and the current modification is by the selected pipe, then the instruction performing the modification must be the last instruction to have entered the pipe containing the AE hazard. In other words, if alpha is small and an alpha register hazard exists, then the short circuit on alpha must occur in the pipe containing the singular AE hazard. If these conditions exist, the AE hazard will not cause a delay.

The AE hazard just described is possible on the LAM, LAC, and LEM instructions. Another delay, closely associated with the AE hazard delay, is encountered when an arithmetic instruction, capable of producing an arithmetic exception condition, finds an LAM, LAC, or LEM instruction at a lower level of any of the four pipes. If found, the arithmetic instruction must wait at level 3 until the LAM, LAC, or LEM instruction has completed its operation (the pipes are clear of any of these three instructions).

Hardware solution: Any LAM, LAC, LEM, or arithmetic instruction capable of producing an arithmetic exception condition will insert an AE hazard bit into the pipe selected by the said instruction. This bit travels with the instruction as it moves down the pipeline. Any subsequent LAM, LAC, or LEM instruction waits at level 3 until only one or none of the AE hazard bits exist in the four pipes. If

only one pipe contains an AE hazard, then that pipe is selected according to the previously stated conditions. If no AE hazards exist, then the pipe is selected according to the known scalar routing rules based on register hazards, short circuits, and X, Y, Z buffer activity.

Also, an instruction which has the potential of an arithmetic exception condition will make a test to see if any LAM or LAC indicator bits are below level 3. If so, then the AE possible instruction waits at level 3 until the LAM or LAC instruction has been cleared from the pipe.

Delay avoidance: Insert other non-AE hazard instructions in front of LAM, LAC, or LEM instructions to allow time for the AE hazard to clear. Also, perform nonarithmetic operations after LAM or LAC instructions to allow time for the LAM or LAC to clear the pipe.

Delay time: AE hazard delay time amounts to four clocks plus AU time plus memory time if the AE hazard producing instruction immediately preceded the LAM, LAC, or LEM instruction.

2. Store Program Status Hazard

DELAY

Before a Store Program Status (SPS) instruction can leave level 3, it must have the final result code and condition code for instructions prior to the SPS. Of the four fields stored by the SPS (CP memory usage, BSR, CC, and RC), only the CC and RC can be modified while the SPS is in level 3. CP memory usage does not change during program execution, and the BSR field can only change while an Execute instruction is in level 3. The SPS waits until all result code or condition code modifying instructions have cleared all four pipes. It then proceeds down the pipe selected according to the SPS storage address.

Hardware solution: An SPS instruction makes a test for a signal called hex register hazard in the level 3 controller. This signal will be true if any result code or condition code modifying instructions are in any of the pipes below level 3. When the last instruction to set the result code or to set the condition code (prior to the SPS) has cleared its selected pipe, then the SPS is released from level 3.

Delay avoidance: There is no easy way to avoid a hex register hazard since nearly all instructions either modify the result code or the condition code. The instructions remaining after the RC and CC modifying types are removed constitute only the special operation type instructions: LEM, LAM, LAC, LLA, XCH, LF, LFM, STF, STFM, SPS, LEA,

MCP, MCW, INT, PSH, PUL, MOD, BLB, BLX, FORK, JOIN, BCC, BRC, and BAE. Nearly every one of these has its own special delay except for LLA, LEA, and INT. Very little programming can be done using these remaining three instructions.

Delay time: SPS hazard delay time is dependent upon the time for the last result code or condition code modifying instruction to clear the pipe. If this instruction immediately precedes the SPS, then the delay time is four clocks plus AU time plus memory time, providing the AU output does not become blocked due to multiple AU outputs destined for the register file. AU time is from Table 1. Memory time is eight clocks if the hex register hazard producing instruction makes a memory request. It is zero if no request is made.

E. Branch Hazards

1. Result Code Hazard

DELAY

This hazard occurs when a "Branch on Result Code" (BRC) instruction arrives at level 3 for execution and the latest result code value has not been set yet. The latest result code modifying instruction is below level 3 but has not cleared the pipe. The result code will be set simultaneously with the AU result being placed in the register file. A BRC instruction waits in level 3 until the last instruction to modify the result code has cleared the pipe. There may, at this time, still be other prior result code setting instructions in other pipes, but these result codes were evidently not needed and, in fact, will never affect the result code if some other result code modifying instruction came later and was the last one before a BRC instruction.

Unconditional branch instructions (branches with an R-field of 7) do not experience this delay; they simply make their branch address request upon arrival at level 3. A conditional branch instruction within the top four words of an octet will make a request for its branch address on the assumption that the branch will be taken. This feature employs the "dual look-ahead" hardware of the IPU which is based on the concept that the branch octet of instructions will be available in one of the instruction buffers (either KA or KB) by the time the branch decision is made.

In the event that the branch is not taken, then the fact that at least four instructions along the downstream path still reside in the current instruction buffer allows the normal look-ahead octet of instructions to be refetched while the four remaining downstream instructions are being processed.

Hardware solution: Branch hazards are monitored by examining one of three signals, depending upon the type of branch instruction. The three types of branch instructions are:

- BRC Branch on result code
- BCC Branch on condition code
- BAE Branch on arithmetic exception

These three branch instructions each check for their own type of hazard:

- BRC checks for result code hazards
- BCC checks for condition code hazards
- BAE checks for arithmetic exception hazards

The register stack contains, as one of its components, three columns of bits, one for each type of hazard. These bits track the instructions down the pipe. When all bits in a particular column have cleared, then that hazard associated with that column has cleared. The branch decision can be made when the hazard associated with that branch has cleared.

Delay avoidance: For branch on compare code instructions, result code setting instructions can be inserted between the branch (BCC) and the compare code setting instruction. In this instance, the inserted instructions may be selected from a wide variety of the CP instruction set. This is not quite so true with BRC or BAE instructions since they have the characteristic of waiting on result producing instructions. It is hard to find more than one compare code setting instruction that can be inserted between a BRC or BAE and the last result code or AE setting instruction.

Delay time: Branch hazard delay time is dependent upon the time for the last RC, CC, or AE setting instruction to clear the pipe for a BRC, BCC, or BAE instruction, respectively. If the hazard causing instruction immediately precedes the BRC, BCC, or BAE instruction, then the delay time is four clocks plus AU time plus memory time, providing the AU output does not become blocked due to multiple AU results destined for the register file. AU time is from Table 1. Memory time is eight clocks if the hazard-causing instruction makes a memory read request. It is zero if no request is made.

2. Condition Code Hazard

DELAY

This hazard occurs when a "Branch on Condition Code" (BCC) instruction arrives at level 3 for execution and the latest condition code value has not been set yet. This hazard is similar to the Result Code Hazard just described in section E.1.

3. Arithmetic Exception Branch Hazard

DELAY

This hazard occurs when a "Branch on Arithmetic Exception" (BAE) instruction arrives at level 3 for execution and the latest arithmetic exception condition code has not been set yet. This hazard is similar to the Result Code Hazard just described in section E.1.

II. Scalar-Vector Hazards

A. Vector Parameter File Modification

DELAY

Before a vector instruction can transmit the vector parameters to the Memory Buffer Unit (MBU), it must have the final values in the vector parameter file (VPF). The VPF consists of the lower eight registers of the forty-eight-word register file (words 27 through 2F hex). This file may be acquired from memory (VECTL), or it may be the current contents of the VPF. In either case there must not be any scalar instructions currently in the CP pipeline that will modify the vector parameter file registers. If any such VPF modifying instruction exists below level 3, then the vector instruction at level 3 will wait until the VPF hazard clears.

Hardware solution: A vector instruction at level 3 tests a signal called "any V haz" to determine whether any register of the vector parameter file will be modified by an instruction below level 3. The "any V haz" signal is made from an ORing of four columns of bits in the register stack. These bits track the instruction as it moves down the CP pipeline. The top bit of this column is set when an instruction with an index or vector register destination moves from level 3 to 4. The column splits into four columns at level 5, at the point where four pipes begin.

From the statement describing the setting of the top most bit, it is apparent that the signal "any V haz" also detects any index register modifying instructions. This protects

against picking up a wrong index value when the starting indices are added to the vector starting addresses as part of the vector initialization process.

Delay avoidance: Avoid modifying index registers or vector parameter registers immediately before a vector. Insert other arithmetic or base register modifying instructions before vectors.

Delay time: Vector parameter file hazards are dependent upon the pipe time of the index or vector modifying instruction. If the VPF destination instruction immediately precedes the vector, then the delay time is four clocks plus AU time plus memory time, providing the AU output does not become blocked due to multiple AU results destined for the register file. AU time is from Table 1, and memory time is eight clocks if the scalar instruction makes a memory read request.

D. Scalars Writing Over Vector Input Arrays

FUNCTIONAL

There is no hardware checking to guard against scalar store instructions writing over vector read data arrays. For this functional hazard to exist, the following conditions must be true:

- (a) The CP must have been in the fork mode when the scalar store was executed and must remain in that mode until a first vector after the scalar store is executed with the "allow current" bit set to "one."
- (b) The first vector must closely follow the scalar store type instruction. Store types are all stores, push, pull, modify, or exchange instructions.
- (c) The scalar store type instruction must write into an area of memory that is used for the initial input data by the vector. Therefore

Therefore, in order to not encounter this type of scalar-vector hazard, complement the sense of any condition a, b, or c above. In particular, the simplest method of preventing the acquisition of old data (if condition C is a requirement of the program) is to turn "off" (zero) the "allow current" bit of the first vector after the scalar store.

Hardware solution: Turning "off" (zeroing) the "allow current" bit causes the vector to wait until all pipes have been emptied and all stores currently residing in the Z buffers of the MBU's to be forced into memory. The vector is allowed to make its vector parameter file request while the Z buffers are being emptied.

It is also possible to place the CP in the join mode during execution of the scalar store instruction and then return to the fork mode prior to the vector. If the vector has "allow current" on, then only those pipes executing stores in the join mode will perform a forced write operation to purge their Z buffers of write data. Other non-Z-join pipes are not required to force their data into memory.

Delay avoidance: Negate any of the three conditions a, b, or c above. Preferably, use the join mode to prevent vector read data from being picked up from a given memory area before scalar stores have had time to write into that same memory area.

Delay time: Assume that conditions b and c are program requirements. If this is so, then it is necessary to invoke the join mode to prevent the acquisition of old rather than new data. Now, if the first vector after a scalar store is a VECTL, then the delay time for emptying the joined pipes is overlapped with the load file fetch time for obtaining the vector parameters of the VECTL instruction. The emptying of the joined pipes may also be overlapped with the transmittal of the vector parameters to the MBU. If the joined pipes are not emptied by the time that the vector is ready to request its first read data from memory, then the read requests are held up until the joined pipes are cleared.

If the first vector is a VECT, then the time for emptying the joined pipes can only be overlapped with vector initiali-

zation to the MBU. For this case, vector read requests may be delayed more often, depending upon the time required to clear the joined pipes.

C. Vectors Writing Over Scalar Read Data

FUNCTIONAL

This This hazard is essentially the reverse of hazard II.B just described. In this case a similar set of conditions must be true for the functional hazard to exist:

- (a) The vector must have the "allow following" bit set to "one."
- (b) The first scalar read must closely follow the vector, and there must not have been any intervening join instructions.
- (c) The vector must write into an area of memory from which the scalar read data is obtained.

Therefore, in order to not encounter vector-scalar hazards, complement the sense of any condition a, b, or c above. In particular, the easiest method of preventing the acquisition of old data (if condition c is a requirement of the program) is to zero the "allow following" bit of the vector.

Hardware solution: Turning "off" (zeroing) the "allow following" bit causes the vector to enter a state where it waits for any vectors in progress to complete before executing the next instruction at level 3. Any vector in progress includes the current vector, so it is not possible for a subsequent scalar (except for a load file instruction) to begin execution until the current vector and all preceding vectors have completed. This also includes the completion of all outstanding scalar reads that may be in progress in other pipes during the current vector execution.

Preventing the subsequent scalar from beginning execution guarantees that vector output data will have been written into memory before a scalar read request is issued for the same area of central memory.

Notice that load file scalar requests are allowed in the join mode. This is for the purpose of obtaining a new vector parameter file and then making singleword or halfword modifications before executing a VECT instruction. The memory address of the load file octet is monitored throughout the duration of the vector with "allow following off." Should the vector write over the load file octet location, a flag (alpha hazard flag) will be set to "one." This flag causes the load file octet to be refetched at the completion of the vector.

A hazard that is noncorrectable, along this same line of thought, is when the vector writes over the instruction location of the load file instruction. Should this happen, it is not possible to reconstruct the state of the register file prior to execution of the load file instruction. If the load file instruction location is modified to anything other than a load file to the same register file, then the program will most likely produce erroneous results. Again, the rule "never modify instructions" should prevent most individuals from making this mistake.

Delay avoidance: Negate any of the three conditions a, b, or c preceding. Preferably, turn off the "allow following" bit to prevent reading scalar data before the preceding vector has written into a given area of memory.

Delay time: There is no overlapping of subsequent scalars (except for load file instructions) if the "allow following" bit is "zero." Prior vectors may continue processing in parallel with the last vector if the prior vectors had their "allow following" set to "one." However, the next scalar after the last vector must wait until the longest vector has completed since it waits for all vectors to complete. The longest vector may not necessarily be the last vector. Therefore, the actual delay may exceed the expected delay unless the length of all vectors are taken into account.

If the next instruction after the vector with "allow following" set to "zero" is another vector, then the next vector is allowed to request its vector parameter file and to initialize the MBU but to go no further.

D. Alpha Hazards During Vectors in the Fork
and Join Mode

FUNCTIONAL

DELAY

To begin with, scalar instructions by themselves cannot cause a functional alpha operand hazard. Emphasis here should be placed on the word "functional" as that means a possible error-producing hazard. Alpha operand delays do occur among purely scalar instructions (scalars executed by themselves, clear of vectors), but these delays do not produce errors in results. These delay-generating conditions were described in section B under Central Memory Address Hazards.

Scalar instruction hazards were discussed in sections I.C.1 and I.C.2 for scalars operating clear of vectors and in section I.C.3 for scalars and vectors operating in parallel.

What is of concern in this section is scalar alpha hazards caused by a prior vector. That is, a vector writing over the memory location of a subsequent vector parameter file of a VECTL instruction or a subsequent Load File from memory. Vectors writing over scalar read data was discussed in Section II.C.

Assume that the join mode has been established within some list of scalar instructions prior to the arrival of a first vector at level 3. If the vector is a VECTL, which receives its vector parameter file (VPF) from memory, a test is made for an alpha hazard before issuing the load file request. Since being in the join mode prior to the arrival of this vector guarantees that there are currently no other vectors in progress,

the alpha hazard detection hardware initiates a forced write signal to the MBU containing the hazard-producing store; i.e., the store whose address agrees with the alpha address of the vector instruction. The VPF load request is issued after the forced write has been completed. Delay, in this case, is due to waiting for write data to arrive in memory prior to issuing the read request. Had the alpha hazard not existed, the forced write operation would have taken place as a normal part of the preparation for the vector in the join mode; but the VPF load would not have had to wait on the write to complete.

Now, consider the case where a first vector is executed in the fork mode but has its "allow following" bit set to "zero"; then, suppose a second vector follows immediately. This second vector makes its test for alpha hazards in the "vector plus one" state. Here the situation is different because now it is possible for both vectors and scalars to be in progress simultaneously in any of the four pipes. The alpha hazard logic cannot issue a forced write command to a pipe that is executing a vector because (fortunately) that pipe will simply ignore the command. The logic will, however, issue a forced write to the pipes containing scalar stores; and, in addition, the alpha hazard logic will cause the VPF load request to be delayed until the scalar forced writes are completed. The unprotected case is seen to be when the first vector, with "allow following" off, writes over the vector parameter file (in memory) of the succeeding vector. It is difficult to protect against this case

with hardware since the vector parameter file address that is contained in the AR register of level 3 is erased during vector initialization. That is, register AR becomes involved in the process of transmitting vector parameters to the MBU and cannot continue the function of monitoring a VPF address in AR against all output addresses of the first \vec{C} vector. It is more important, in terms of time that would otherwise be lost on all joined vector initializations, for the initialization to proceed and to sacrifice the alpha hazard hardware checking for this case. Protection of a VPF in memory can be insured by software by using the sequence:

```
SEQ1  VECTL  "AF" = 0
      LF      (load VPF)
      VECT
```

instead of,

```
SEQ2  VECTL  "AF" = 0
      VECTL
```

The VPF of the second VECTL of sequence 2 is unprotected since the second VECTL proceeds through initialization while the first VECTL is still executing. In sequence 1 the alpha address of the LF instruction is checked against all \vec{C} vector addresses of the first VECTL, so the VPF for VECT is protected against modification by VECTL.

Hardware solution: A test is made for scalar alpha hazards prior to making the load file request for a VECTL instruction. This is done regardless of the fork or join mode but is intended

for the join mode. If an alpha hazard, due to a prior scalar storing over a VECTL's alpha address, is seen during the fork mode, then the forced write takes place the same as in the join mode. However, if a prior vector writes over a VECTL's alpha address and the VECTL is executed, all the while remaining in the fork mode, then the alpha hazard will not be seen and the VECTL can pick up its VPF prior to modification by the first vector.

Delay avoidance: Avoid modifying vector parameter files (VPF) of subsequent vectors by means of scalar stores or vector writes (to memory) which are in the immediate vicinity of a vector using the VPF being modified. If separation of modification from use by the method of instruction insertion is not feasible, then at least insert a join instruction between modification and use to prevent functional errors.

Delay time: Needs FUSS prediction.

III. Vector Hazards

A. Vectors Storing Over Their Own Input Data Arrays FUNCTIONAL

This hazard results from a violation of the familiar "Vector Hazard Rule," which states that:

A "hazard condition" occurs whenever the present octet address of input vector \vec{A} or \vec{B} or the next four octet addresses for each of vectors \vec{A} or \vec{B} is the same as the present result octet address or the eight past result octet addresses of output vector \vec{C} .

If the Vector Hazard Rule is violated, the "old" rather than the "new" (updated) information is used as the operand. For example, a vector will use the "old" values for the \vec{B} vector operands if the element address of c_i is one greater than the element address of b_i and all vectors are assigned a positive increment direction during the self-loop. Hardware is not built in to detect this hazard. Also, delay avoidance and delay time descriptions do not apply to functional hazards.

B. Addressing Conflicts Between Two or More
Vectors Executing Simultaneously in Parallel Pipes FUNCTIONAL
Pipes

This addressing conflict is caused by the independence of separate pipes executing in parallel. Unless one is careful, two vectors started in separate but parallel pipes can have their data paths (in memory) cross one another. Input paths crossing input paths cause no trouble. However, let an input data path cross behind an output data path of an earlier vector; and program errors are almost certain. If the memory paths cross like an "X," then the time and phase (ahead or behind) relationship of reaching the intercept point is important. Since vector rates are influenced by memory interference, it is difficult to predict the time of reaching the intercept point for either vector. Therefore, rate control is impossible. So, to prevent memory read-write collisions between vectors, these intercepting vectors must be executed independently of one another.

Another type of intercept would be the parallel path type, particularly when the parallel paths form a single line tracing the same area of memory. In this case the vector rates are also quite critical to program execution. For example, if the output vector were trailing an input vector through contiguous locations, all would be fine (assuming the output vector was the second vector to start execution). Suppose that, subsequently, the output vector of pipe 1 caught up with

and passed the input vector of pipe 2. Now, the first vector would be reading output data from the second vector, a difficult condition to contend with, especially when trying to interpret program results following execution; i.e., man, take a look at that dump! These vectors must also be executed independently of one another.

Hardware is not implemented to detect this hazard. Also, delay avoidance and delay time descriptions do not apply to functional hazards.

C. Halfword Z-fill-in Hazards

FUNCTIONAL

This hazard is similar to the one just described in section III.B since it is caused by two or more vectors executing simultaneously in parallel pipes. An unusual characteristic of this hazard is that it is due to two or more halfword output vectors writing over the same area of memory. It is also quite difficult to predict the hazard based on Fortran compiler algorithms for finding two vectors writing into the same memory space because the errors may arise in halfword locations within the octet being modified and not necessarily at the halfword location common to both vectors.

The hazard is due to the fact that halfword output vectors, which do not fill up an entire octet with halfwords, require a halfword Z-fill-in operation. A Z-fill-in involves both a read and a write cycle, controlled by the Memory Buffer Unit. Since memory only accepts singleword stores (there are eight zone enable lines controlling the word of an octet to be stored), the MBU must recognize halfword stores which do not fill both halves of a singleword location. This is done by examining the sixteen halfword zone bits of the Z buffer of the MBU. Any even-odd bit pair forming a true exclusive or logical combination indicates the need for a Z-fill-in operation.

The operation is as follows:

- (1) The address of the octet to be stored into memory is first sent to memory as a read request for that octet.

- (2) The requested octet is loaded into the ZB buffer of the MBU.
- (3) The "filled" halfwords of the Z buffer are transferred to the ZB buffer. "Unfilled" halfwords are not transferred.
- (4) The "now updated" ZB buffer is written into memory at the storage octet address. Zone enable lines are a logical "one" for those singlewords that have been updated.

A halfword Z-fill-in hazard occurs when the following sequence occurs:

- (1) A first pipe requests an octet for the purpose of Z-fill-in.
- (2) A second pipe requests the same octet for the purpose of Z-fill-in.
- (3) The first pipe modifies one halfword of the octet and then stores the octet back into memory.
- (4) The second pipe modifies some other halfword of the same octet and then stores it back into memory.

Operation 4 above has just erased the work done by the first pipe. Results in the common octet which were to have been stored by the first pipe are lost, and recovery is impossible.

In order to protect against this functional hazard, one must make a complete examination of halfword \vec{C} output vectors, that are running simultaneously in the fork mode, to be sure

that memory octets common to both vectors are not being used. If this is not possible, then vectors with halfword outputs should be run with "allow following" turned off, set to zero.

Hardware is not implemented to detect this hazard. Also, delay avoidance and delay time descriptions do not apply to functional hazards.