

# CONTENTS

<b>CP02 MODULE</b>	<b>1</b>
CP02 General Description .....	1
Module Assembly Components .....	2
<b>ADDRESS AND SCALAR REGISTERS</b>	<b>7</b>
Address Registers .....	7
Error Codes .....	9
A Register Memory References .....	11
Special Register Values .....	11
Scalar Registers .....	13
Instruction Issues .....	13
S Register Memory Reference .....	13
Special Register Values .....	14
Lower/Upper Scalar Register Load .....	14
<b>B AND T REGISTERS</b>	<b>15</b>
<b>ADDRESS/SCALAR ADD</b>	<b>19</b>
<b>SCALAR LOGICAL</b>	<b>21</b>
Address and Scalar Mask .....	23
Transmit <i>nm</i> to <i>Si</i> , <i>Si</i> Upper, <i>Si</i> Lower .....	25
<b>ADDRESS/SCALAR POP/PARITY AND LEADING ZERO</b>	<b>27</b>

**INSTRUCTION ISSUE (continued)**

---

Common Memory Requests .....	181
Shared Resource Requests .....	182
Branch Requests .....	182
Exchange Requests .....	182
Interrupt Requests .....	183
Control Signal Distribution .....	183
Branch Instruction Control .....	186
Conditional Branch Instructions .....	186
Unconditional Branch Instructions .....	186
Issue Control .....	187

**EXCHANGE**

---

**201**

Exchange Process .....	201
Deadstart .....	202
Interrupt Flag Set .....	203
Program Exit .....	203
Exchange Sequence .....	203
Exchange Package Descriptions .....	204

**REAL-TIME CLOCK, PROGRAMMABLE CLOCK INTERRUPT,  
STATUS REGISTER, PERFORMANCE MONITOR**

---

**213**

Real-time Clock .....	213
Programmable Clock .....	214
RTC and PC Instructions .....	215
Performance Monitor .....	215
Performance Monitor Instructions .....	217
Clearing the Performance Counters .....	217
Reading the Performance Monitor .....	217
Performance Monitor Block Diagram .....	218
Status Register .....	218

Cache Hit .....	227
Cache Miss .....	228
Cache Addressing .....	229
Potential Cache Problems .....	229
CH Option .....	230
Scalar Cache Instructions .....	230

## Figures

Figure 1. CP Module Assembly Components .....	2
Figure 2. Option Layout Board 1 .....	3
Figure 3. Option Layout Board 2 .....	4
Figure 4. CPU Block Diagram .....	5
Figure 5. Address and Scalar Register Data Paths .....	8
Figure 6. A/S Control Terms .....	10
Figure 7. Memory to A/S-register Block Diagram .....	12
Figure 8. B and T Register Inputs and Outputs .....	15
Figure 9. B/T-register-to-memory Block Diagram .....	17
Figure 10. Carry Bit and Enable Bit Fanouts .....	20
Figure 11. Address/Scalar Logical Block Diagram (Instructions 044ijk through 051ijk) .....	21
Figure 12. Scalar Mask Block Diagram .....	24
Figure 13. A/S Population/Parity/Leading Zero Count .....	29
Figure 14. Address Register Shift .....	32
Figure 15. Shift Count Breakdown .....	34
Figure 16. Address Register Left Single Shift .....	35
Figure 17. Address Register Right Single Shift .....	36
Figure 18. Address Register Left Double Shift .....	37
Figure 19. Address Register Right Double Shift .....	38
Figure 20. Example of an A Register Left Single-shift Instruction .....	39
Figure 21. Example of an Address Register Left Double-shift Instruction .....	41
Figure 22. Example of an Address Register Right Double-shift Instruction .....	42
Figure 23. Scalar Shift .....	44
Figure 24. Shift Count Breakdown .....	46

## Figures (continued)

---

Figure 25.	Scalar Left Single Shift .....	47
Figure 26.	Scalar Right Single Shift .....	48
Figure 27.	Scalar Left Double Shift .....	49
Figure 28.	Scalar Right Double Shift .....	50
Figure 29.	Example of Scalar Left Single-shift Instruction ....	51
Figure 30.	Example of a Scalar Register Left Double-shift Instruction .....	53
Figure 31.	Example of a Scalar Register Right Double-shift Instruction .....	54
Figure 32.	AN Option .....	56
Figure 33.	C90 Operation Mode .....	60
Figure 34.	AM Option Inputs .....	61
Figure 35.	Write Data Path .....	69
Figure 36.	Read Data Path for Pipe 0 (Even Elements) .....	71
Figure 37.	Read Data Path for Pipe 1 (Odd Elements) .....	72
Figure 38.	Vector Register Write Block Diagram (Pipe 0) .....	73
Figure 39.	Vectors 0 through 3 Pipe 0/1 Read Data Path .....	75
Figure 40.	Vectors 4 through 7 Pipe 0/1 Read Data Path .....	77
Figure 41.	Vectors 0 through 3 Pipe 0/1 Write Data Path .....	79
Figure 42.	Vectors 4 through 7 Pipe 0/1 Write Data Path .....	81
Figure 43.	Vector Register Decode Bit Fanout (Pipe 0 and 1 Path 1 Only) .....	83
Figure 44.	Vector Register Decode Bit Fanout (Pipe 0 and 1 Path 2 Only) .....	85
Figure 45.	S Register to Vectors .....	87
Figure 46.	Memory Data to Vectors (Even Elements) .....	89
Figure 47.	Memory Data to Vectors (Odd Elements) .....	91
Figure 48.	Vector Logical Block Diagram .....	94
Figure 49.	Vector Merge Operation .....	96
Figure 50.	1750j0 Instructions .....	98
Figure 51.	Function of the 175ij4 Instructions .....	99
Figure 52.	Iota Pipe 0 .....	100
Figure 53.	Function of the 070ij1 Instructions .....	101
Figure 54.	Vector Add Block Diagram .....	105
Figure 55.	Shift Count Breakdown .....	108
Figure 56.	Vector Shift Block Diagram .....	109

## Figures (continued)

---

Figure 57.	Vector Right Shift .....	110
Figure 58.	Vector Right Double Shift .....	111
Figure 59.	Vector Transfer .....	112
Figure 60.	Vector Compress .....	112
Figure 61.	Vector Expand .....	113
Figure 62.	Vector Population/Parity/Leading Zero Block Diagram .....	116
Figure 63.	Floating-point Add .....	122
Figure 64.	Floating-point Add Sticky Bits .....	123
Figure 65.	Floating-point Format .....	123
Figure 66.	Floating-point Add Flowchart .....	129
Figure 67.	Newton's Method for Approximating Roots .....	132
Figure 68.	Reciprocal Approximation Functional Unit .....	140
Figure 69.	Floating-point Multiply Block Diagram .....	149
Figure 70.	Floating-point Multiply First-level Summation ....	151
Figure 71.	Vector Storage of Bit Matrices .....	154
Figure 72.	Mathematical Representation of Matrices A and B ..	155
Figure 73.	B Matrix and B <sup>t</sup> Matrix Relationships .....	155
Figure 74.	Multiplication of A and B <sup>t</sup> .....	156
Figure 75.	Bit Matrix Multiply Block Diagram Pipe 0 .....	159
Figure 76.	Bit Matrix Multiply Block Diagram Pipe 1 .....	161
Figure 77.	IC Options Bit Layout .....	166
Figure 78.	IC Block Diagram .....	167
Figure 79.	IC Option Terms .....	168
Figure 80.	Memory-to-Instruction Buffers (Path 1) .....	169
Figure 81.	Memory-to-Instruction Buffers (Path 2) .....	170
Figure 82.	Common Memory Path Code 1 Fanouts .....	171
Figure 83.	Common Memory Path Code 2 Fanouts .....	173
Figure 84.	Instruction Issue Block Diagram .....	175
Figure 85.	Format for a 1-parcel Instruction .....	176
Figure 86.	Format for a 3-parcel Instruction .....	176
Figure 87.	Format for a 4-parcel Instruction .....	177
Figure 88.	B <sub>jk</sub> (Exchange P) Fan-out Bits .....	188
Figure 89.	JA-to-IC Parcel Data for Branches .....	189
Figure 90.	Path 1 CH to IC to JA Option .....	190
Figure 91.	Path 2 CH to IC to JA Option .....	191
Figure 92.	JA Option Block Diagram .....	193

## Figures (continued)

---

Figure 93.	Instruction Data Distribution A/S/B/T Registers	195
Figure 94.	CIP Distribution	196
Figure 95.	CIP Distribution to HF Option	197
Figure 96.	Instruction Data Distribution to VA and BT Options	198
Figure 97.	CIP Distribution	199
Figure 98.	CIP Distribution to HF Option	200
Figure 99.	Exchange Package	206
Figure 100.	RTC and PCI Block Diagram	214
Figure 101.	Performance Monitor Block Diagram	219
Figure 102.	Status Registers	221
Figure 103.	Cache Layout	228
Figure 104.	Memory Addresses	229

## Tables

---

Table 1.	A/S Register Entry Codes	9
Table 2.	B/T Register Instructions	16
Table 3.	A/S Adder Instructions	19
Table 4.	Scalar Logical Functional Unit Instructions	22
Table 5.	Address Logical Functional Unit Instructions	23
Table 6.	Scalar Mask Instructions	23
Table 7.	Address Mask Instructions	24
Table 8.	Transmit <i>nm</i> to <i>Si</i> Instructions	25
Table 9.	Scalar Pop Count/Parity and Leading Zero Count Instructions	28
Table 10.	Address Register Shift Instructions	31
Table 11.	Scalar Shift Instructions	43
Table 12.	Recode Groups	56
Table 13.	Vector Register Options	64
Table 14.	VM/VR Data Steering	68
Table 15.	Vector Logical Instructions	95
Table 16.	Vector Merge Instructions	95
Table 17.	Vector Mask Operations	97
Table 18.	Vector Mask Test Operations	98
Table 19.	Iota Instruction	99
Table 20.	Vector Add Instructions	103
Table 21.	Vector Shift Instructions	107

## Tables (continued)

---

Table 22.	Vector Population/Parity Instructions .....	118
Table 23.	Floating-point Add Functional Unit Instructions ...	123
Table 24.	Reciprocal Approximation Values .....	135
Table 25.	Floating-point Reciprocal Approximation Instructions .....	139
Table 26.	Floating-point Multiply Functional Unit Instructions	146
Table 27.	Bit Matrix Multiply Instructions .....	157
Table 28.	IC Options .....	163
Table 29.	Read-out Path Codes .....	179
Table 30.	Interrupt Modes Register Bit Assignments .....	208
Table 31.	Flag Register Bit Assignments .....	209
Table 32.	LAT Fields .....	211
Table 33.	RTC and PC Instructions .....	215
Table 34.	Performance Monitor .....	216
Table 35.	Performance Monitor Instructions .....	217
Table 36.	Status Register (SR0) .....	222
Table 37.	Status Register 4 (SR4) .....	223
Table 38.	Destination Codes .....	223
Table 39.	Status Register 7 Bit Definitions .....	224
Table 40.	Register Parity Error Code .....	224
Table 41.	CH Option Bits .....	230
Table 42.	Scalar Cache Instructions .....	230

○

Handwritten text, mostly illegible due to fading and bleed-through. Some words like "The" and "of" are faintly visible.

○

Handwritten text, mostly illegible due to fading and bleed-through. Some words like "The" and "of" are faintly visible.

○







# CP02 MODULE

## CP02 General Description

---

The CP02 module contains the central processing unit (CPU) for the CRAY T90 series computer systems. There is one CPU per CP02 module. The CRAY T90 series CPU is compatible with the CRAY C90 series CPU. This means that code compiled on the CRAY C90 series system will run on a CRAY T90 series system.

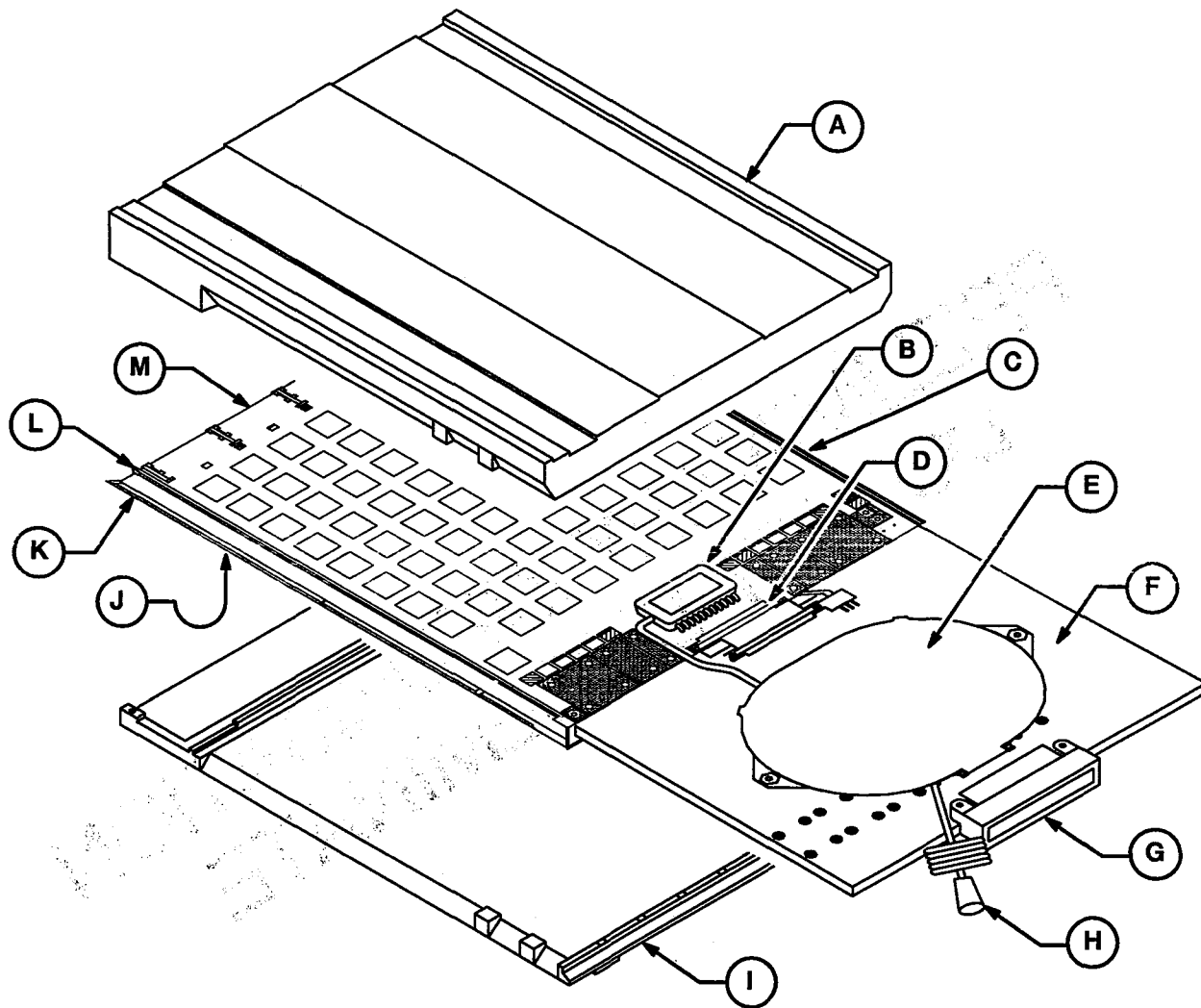
There have been many enhancements to the CRAY T90 series CPU and several new instructions added to increase the performance. Figure 1 illustrates CP module components. Figure 2 and Figure 3 show the basic functions and locations of all options on a CP module. Figure 4 shows a block diagram of the CPU.

The CP modules are arranged in stacks in the system. A CRAY T94 system contains one stack of as many as four modules. A CRAY T916 system contains up to two stacks of as many as eight modules. A CRAY T932 system contains up to four stacks of as many as 8 modules.

Each module in a stack is independent of the other CP modules in the stack; there are no interconnections between modules in a stack. The CP modules connect directly with either the memory modules, as in the CRAY T94 system, or with the system interconnect board (SIB), as in larger systems.

## Module Assembly Components

Refer to Figure 1 for an illustration of the CP module assembly components. This illustration is provided to show the basic components that are part of all mainframe modules. Sizes of various components differ between modules.



- |   |                                     |   |                     |
|---|-------------------------------------|---|---------------------|
| A | Flow Block, Board 1                 | H | Fiber-optic Coupler |
| B | Optical Receiver                    | I | Flow Block, Board 2 |
| C | PC Board Edge Shim                  | J | PC Logic Board 2    |
| D | Maintenance Connector Flex Assembly | K | Outer Rail          |
| E | Fiber-optic Spool Assembly          | L | Inner Rail          |
| F | Voltage Regulator Board Assembly    | M | PC Logic Board 1    |
| G | Maintenance Connector               |   |                     |

Figure 1. CP Module Assembly Components

HB000 I/O Control								
	NA000 Fit Mult	RC000 Recip	TZ000 Clock	HM000 Logic Monitor	MZ000 BS Fanout	TW010 Not Used	RC001 Recip	NA001 Fit Mult
TW000 Not Used	NC000 Fit Mult	RB000 Recip	FA000 Fit Add Coeff	TW006 Not Used	FA001 Fit Add Coeff	OA002 BMM and Parity	RB001 Recip	NC001 Fit Mult
TW002 Not Used	VM007 Vector even R Bit 60-63 W Bit 56-63	AU000 A/S reg Bits 48-55	VM006 Vector even R Bit 52-55 W Bit 48-55	SS000 Shift Pop LZ	OA000 BMM and Parity	OA001 BMM and Parity	VM014 Vector odd R Bit 52-55 W Bit 48-55	VM015 Vector odd R Bit 60-63 W Bit 56-63
HD000 CIP Exchange Package	VM005 Vector even R Bit 44-47 W Bit 40-47	AT000 A/S reg Bits 32-39	VM004 Vector even R Bit 36-39 W Bit 32-39	JA000 Issue Control	VA000 Vector Control	CG000 Checkbit Generation	VM012 Vector odd R Bit 36-39 W Bit 32-39	VM013 Vector odd R Bit 44-47 W Bit 40-47
VF000 Vector Control	VM003 Vector even R Bit 28-31 W Bit 24-31	AS001 A/S reg Bits 16-23	VM002 Vector even R Bit 20-23 W Bit 16-23	BT000 B/T/P reg Bits 0-15 Bits 32-47	CD000 Ports E Cache HIT	CB000 Ports C	VM010 Vector odd R Bit 20-23 W Bit 16-23	VM011 Vector odd R Bit 28-31 W Bit 24-31
TW004 Not Used	VM001 Vector even R Bit 12-15 W Bit 8-15	AR000 A/S reg Bits 0-7	VM000 Vector even R Bit 4-7 W Bit 0-7	CH010 Data Mux Cache 20-23 52-55	CH008 Data Mux Cache 16-19 48-51	CA000 Ports A, A'	VM008 Vector odd R Bit 4-7 W Bit 0-7	VM009 Vector odd R Bit 12-15 W Bit 8-15
HA000 I/O to Mem SBCDBD	CC000 Ports D	IC000 Inst Buffers Bit 0-7 Bit 32-39	CH002 Data Mux Cache 4-7 36-39	CH014 Data Mux Cache 28-31 60-63	CH012 Data Mux Cache 24-27 56-59	CH000 Data Mux Cache 0-3 32-35	IC002 Inst Buffers Bit 16-23 Bit 48-55	VF002 Vector Control
HA002 I/O to Mem SBCDBD	CF004 Write data conflicts	CF000 Write data conflicts	CK000 Data steering	CH006 Data Mux Cache 12-15 44-47	CH004 Data Mux Cache 8-11 40-43	CK002 Data steering	CF002 Write data conflicts	TW008 Not Used
HG000 Maint. Channel	CI000 Section Driver Section 0	CJ000 Section Receiver Section 0	CI004 Section Driver Section 4	CJ004 Section Receiver Section 4	CI002 Section Driver Section 2	CJ002 Section Receiver Section 2	CI006 Section Driver Section 6	CJ006 Section Receiver Section 6
<div style="display: flex; justify-content: space-around; align-items: center;"> <div style="border: 1px solid black; padding: 2px 10px;">ZB008</div> <div style="border: 1px solid black; padding: 2px 10px;">ZB000</div> <div style="border: 1px solid black; padding: 2px 10px;">ZB004</div> <div style="border: 1px solid black; padding: 2px 10px;">ZB002</div> <div style="border: 1px solid black; padding: 2px 10px;">ZB006</div> </div>								

Figure 2. Option Layout Board 1

								HC000 I/O relay data
ND001 Fit Mult	AM001 Integer Multi	TW011 Not Used		HM001 Logic Monitor		AM002 Integer Multi	ND000 Fit Mult	
NB001 Fit Mult	RA001 Recip	OA005 BMM and Parity	FB001 Fit Add Exponent	TW007 Not Used	FB000 Fit Add Exponent	RA000 Recip	NB000 Fit Mult	TW001 Not Used
VR015 Vector 7 Odd Bits 56-59	VR014 Vector 6 Odd Bits 48-51	OA004 BMM and Parity	OA003 BMM and Parity	VS000 Vector Shift	VR006 Vector 6 Even Bits 48-51	AU001 A/S reg Bits 56-63	VR007 Vector 7 Even Bits 56-59	TW003 Not Used
VR013 Vector 5 Odd Bits 40-43	VR012 Vector 4 Odd Bits 32-35	CG001 Checkbit Generation	VA001 Vector Control	JA001 Issue Control	VR004 Vector 4 Even Bits 32-35	AT001 A/S reg Bits 40-47	VR005 Vector 5 Even Bits 40-43	HD001 CIP Exchange Package
VR011 Vector 3 Odd Bits 24-27	VR010 Vector 2 Odd Bits 16-19	CB001 Port C'	CD001 Port E Cache Control	BT001 B/T/P reg Bits 16-31 Bits 48-63	VR002 Vector 2 Even Bits 16-19	AS002 A/S reg Bits 24-31	VR003 Vector 3 Even Bits 24-27	VF001 Vector Control
VR009 Vector 1 Odd Bits 8-11	VR008 Vector 0 Odd Bits 0-3	CA001 Port B, B'	CH009 Data Mux Cache 16-19 48-51	CH011 Data Mux Cache 20-23 52-55	VR000 Vector 0 Even Bits 0-3	AS000 A/S reg Bits 8-15	VR001 Vector 1 Even Bits 8-11	AN000 Address Multi
VF003 Vector Control	IC003 Inst Buffers Bit 24-31 Bit 56-63	CH001 Data Mux Cache 0-3 32-35	CH013 Data Mux Cache 24-27 56-59	CH015 Data Mux Cache 28-31 60-63	CH003 Data Mux Cache 4-7 36-39	IC001 Inst Buffers Bit 8-15 Bit 40-47	TW005 Not Used	HA001 I/O SECEDED
TW009 Not Used	CF003 Write data conflicts	CK003 Data Steering Cache Control	CH005 Data Mux Cache 8-11 40-43	CH007 Data Mux Cache 12-15 44-47	CK001 Data Steering Cache Control	CF001 Write data conflicts	CF005 Write data conflicts	HA003 Maint. Channel
CI007 Section Driver Section 7	CJ007 Section Receiver Section 7	CI003 Section Driver Section 3	CJ003 Section Receiver Section 3	CI005 Section Driver Section 5	CJ005 Section Receiver Section 5	CI001 Section Driver Section 1	CJ001 Section Receiver Section 1	HF000 Perf Monitor
<div style="display: flex; justify-content: space-around; align-items: center;"> <div style="border: 1px solid black; padding: 2px 10px;">ZB007</div> <div style="border: 1px solid black; padding: 2px 10px;">ZB003</div> <div style="border: 1px solid black; padding: 2px 10px;">ZB005</div> <div style="border: 1px solid black; padding: 2px 10px;">ZB001</div> <div style="border: 1px solid black; padding: 2px 10px;">ZB009</div> </div>								

Figure 3. Option Layout Board 2

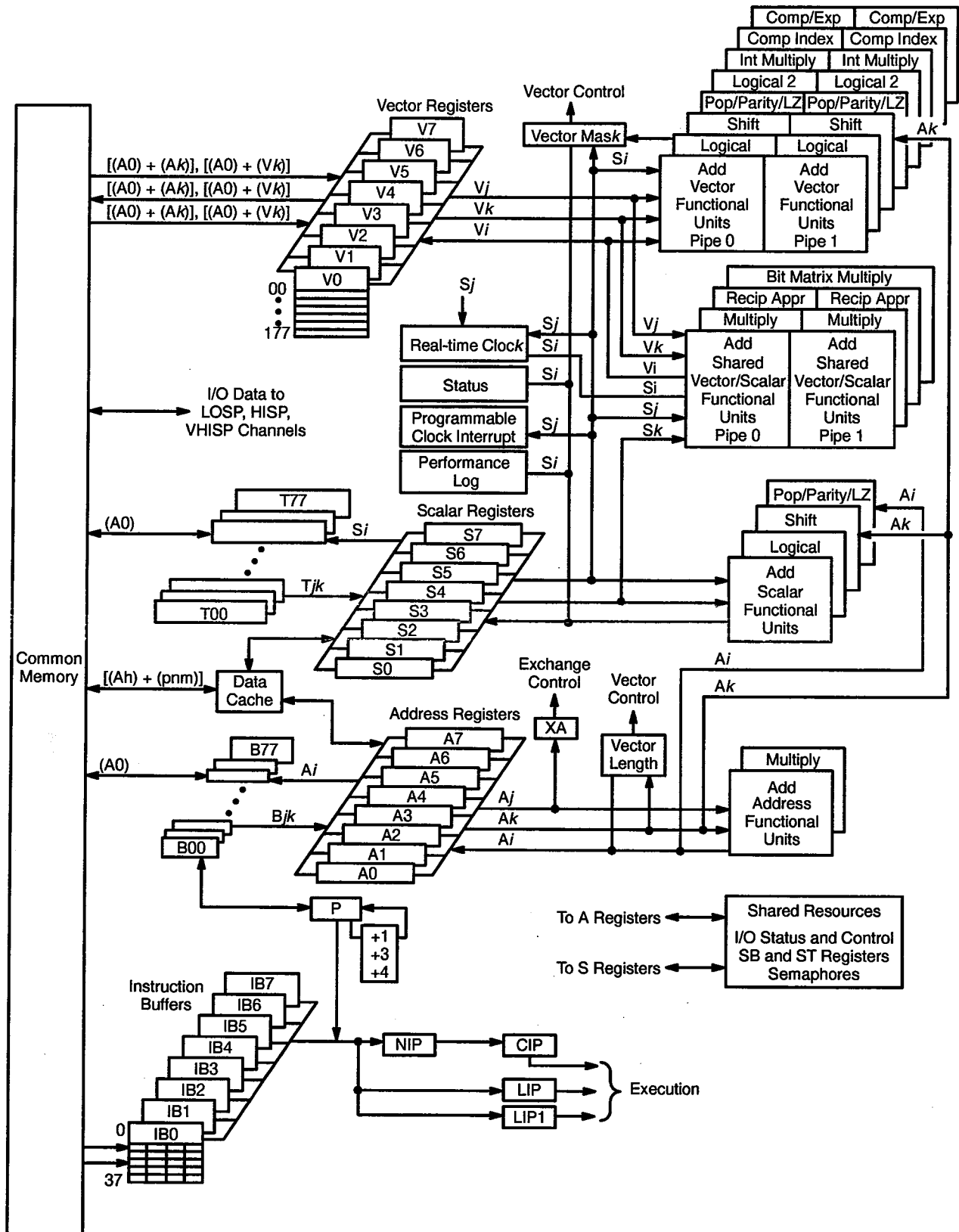
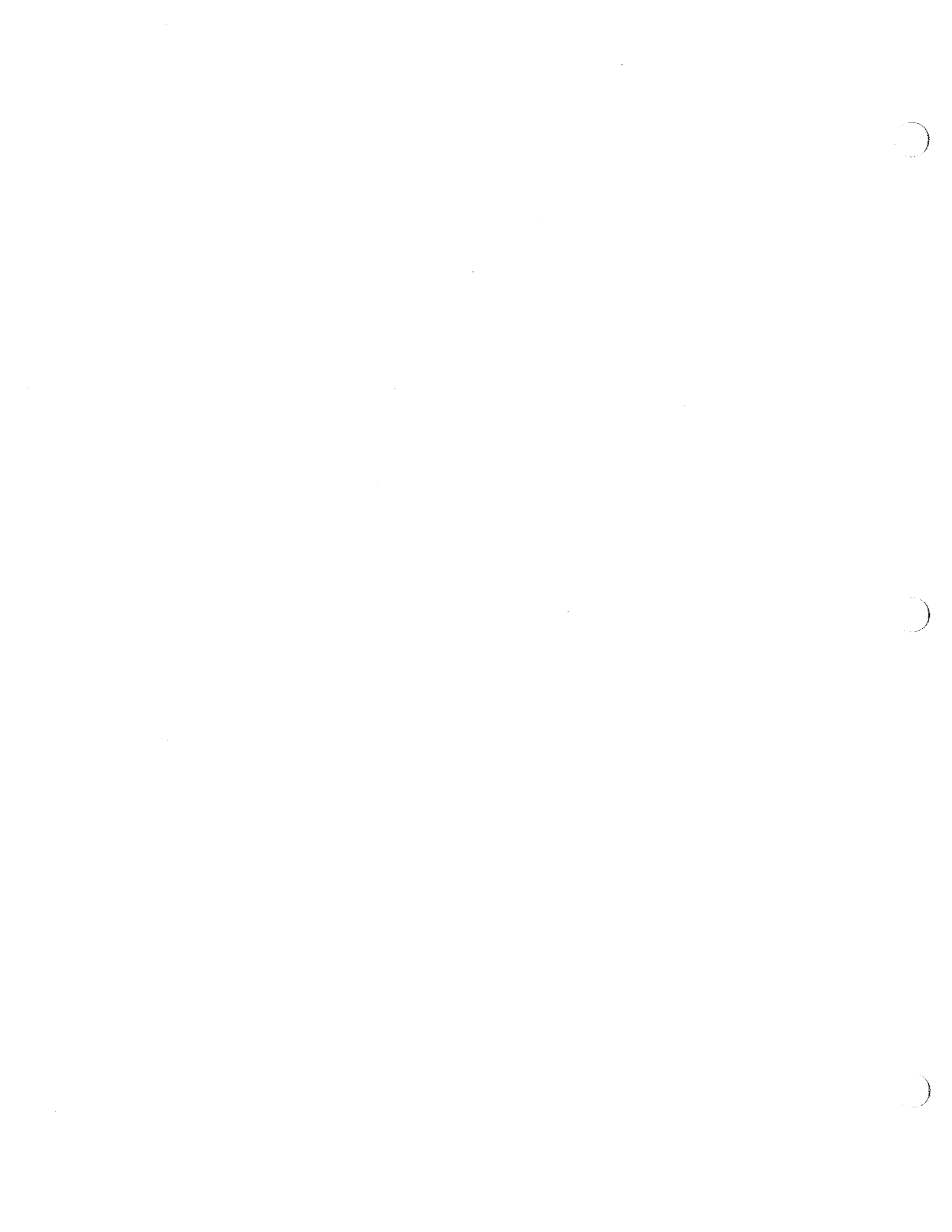


Figure 4. CPU Block Diagram





# ADDRESS AND SCALAR REGISTERS

The address and scalar registers are located on the same options. The following subsections describe the address and scalar registers.

## Address Registers

---

The address and scalar registers are contained on eight options: one AR option, three AS options, two AT options, and two AU options. Each CRAY T90 series CPU contains eight address registers designated A0 through A7. Each register is 64 bits wide (32 bits in C90 mode) and performs the following functions:

- Determines addresses for memory references
- Provides memory reference indexing
- Provides loop control
- Determines shift counts
- Provides I/O channel set-up
- Determines I/O channel status
- Receives results from scalar leading zero and pop count
- Determines vector length
- Provides an exchange address (monitor mode only)
- Provides an index for shared registers and B and T instructions
- Provides operands and results for address add and address multiply
- Transfers data to and from scalar registers
- Provides integer-to-floating-point conversion

As shown in Figure 5, the AR000, AS000, AS001, AS002, AT000, AT001, AU000, and AU001 options each contain an 8-bit slice of the address registers. Figure 5 also illustrates the input and output data paths for the address and scalar registers.

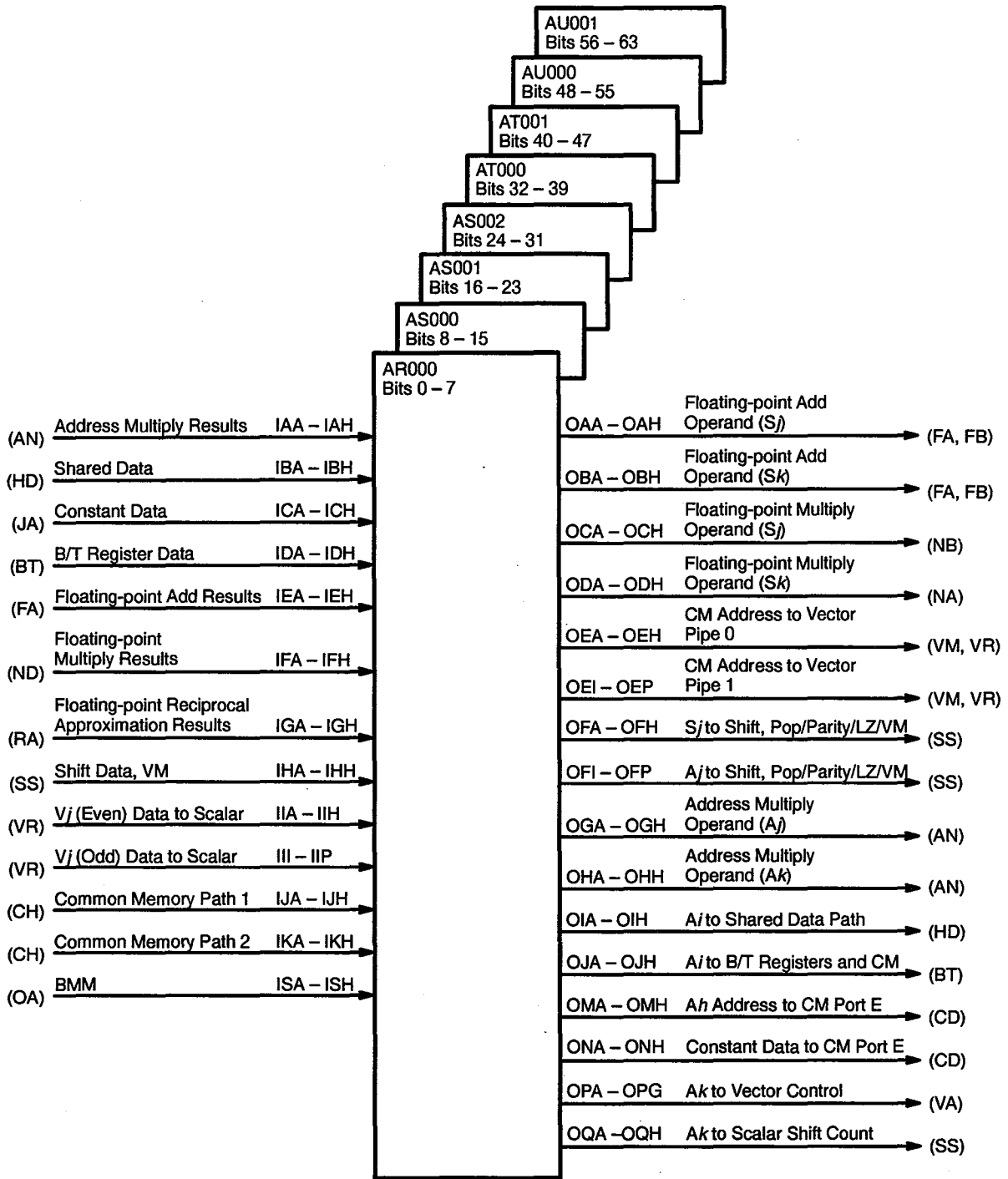


Figure 5. Address and Scalar Register Data Paths

## Entry Codes

---

As part of the instruction decode on the JA option, the JA option sends an A/S entry code to the A/S register options; this code generates the control necessary to complete the operations. The operand data is then transmitted to the appropriate resources, and a destination delay chain is entered on the option. Refer to Table 1 for the address/scalar (A/S) register entry codes and to Figure 6 for an illustration of the A/S control terms.

Table 1. A/S Register Entry Codes

Entry Code	Instruction
0	020 <i>i</i> Constants
1	023 <i>ij</i> 0 <i>Sj</i>
2	023 <i>ij</i> 1 VL data
3	024 <i>ijk</i> B data
4	030,031 <i>ijk</i> Add
5	026 <i>ij</i> (0 – 3), 027 <i>ij</i> (0 – 1) pop/par/lz
6	032 <i>ijk</i> A multiply
7	022 <i>ijk</i> , 04 (2 – 3) <i>jk</i> /mask data
10	N/A
11	073 <i>i</i> (2 – 3) 0 VM data
12	N/A
13	N/A
14	04 (4 – 7) <i>ijk</i> , 05 (0 – 1) <i>ijk</i> Logical
15	N/A
16	05 (2 – 5) <i>ijk</i> , 05 (6 – 7) <i>ijk</i> Shift
17	N/A

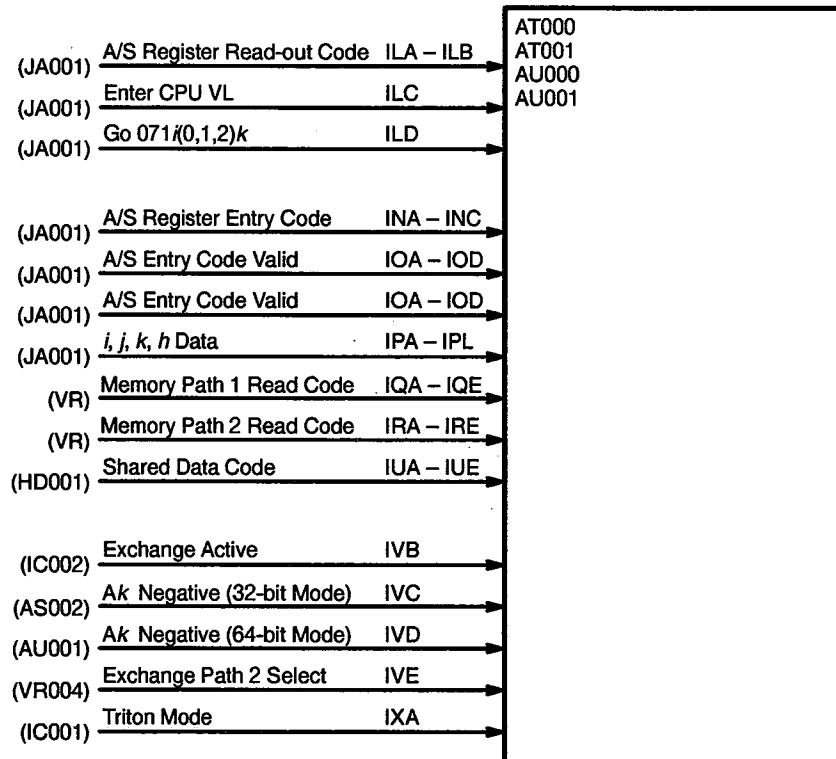
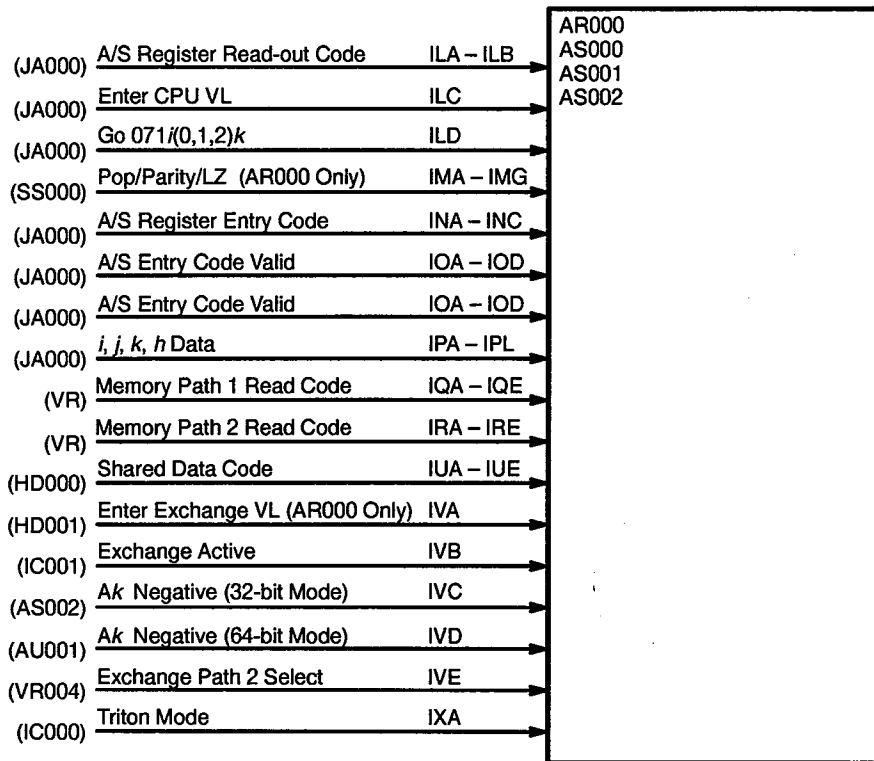


Figure 6. A/S Control Terms

## A Register Memory References

---

Refer to Figure 7 for an A/S-register-to-memory block diagram. The address registers write or read 1 word of memory per instruction. The B registers provide intermediate storage for the address registers. B registers perform memory block references that enable a group of operands to be read from memory with one instruction. These operands are then used by the A registers to generate results that are sent to the B registers and block-stored to memory. Using the B registers as buffer storage is advantageous because it takes fewer clock periods to do a block reference than to issue several individual address or scalar references.

The A registers also have an access path to cache memory. This provides access to common memory data without having to reference memory directly. If the requested address resides in cache, a *cache hit* is initiated and the data is read from cache memory instead of common memory.

## Special Register Values

---

The A0 register has special features that the other A registers do not have. The A0 register holds the starting address for all block transfers for the B, T, and V registers and branch control. A0 is the only register that can be tested for equal-to-zero, not-equal-to-zero, positive, or negative conditions using A0 conditional branch instructions. This register also has a special feature for reading data.

If A0 is specified as an operand in the *h*, *j*, or *k* field of an instruction, it will not send the actual contents of the register. Instead, the register sends a value of 0 if A0 is used in the *j* or *h* field, or it sends a value of 1 if A0 is used in the *k* field. If A0 is used in the *i* field, the actual contents of the A0 register are sent.

Because the A registers in this system are now 64 bits wide, special Triton mode instructions have been implemented. These instructions are part of the extended instruction set (EIS). These instructions make the A registers functionally equal to S registers and enable A registers to be shifted and logical operations to be performed. To execute these instructions, an EIS 005400 instruction must precede the actual A register instruction. If a Triton mode instruction is issued while the machine is in C90 mode, the results of the operation are undefined.

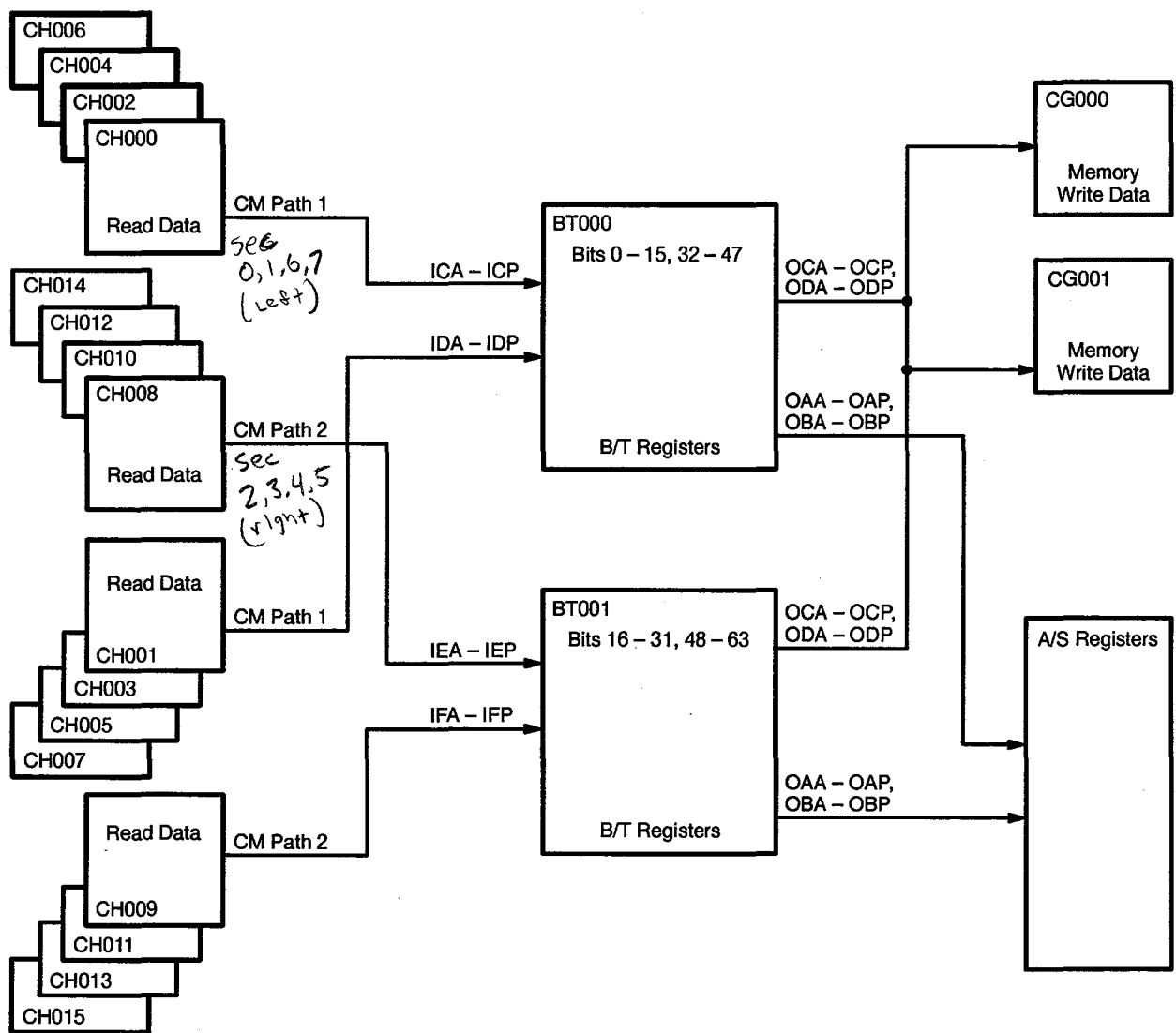


Figure 7. Memory to A/S-register Block Diagram

## Scalar Registers

---

The CPU contains eight scalar registers that are designated S0 through S7 and are 64 bits in length. The scalar registers are contained on the AR, AS, AT, and AU options (refer again to Figure 5).

The scalar registers send operands to, and get results from, the scalar functional units and the floating-point functional units. The functional units perform integer and floating-point arithmetic as well as logical operations. The scalar registers read and write central memory through the T registers and also read and write the data cache. In addition, there are paths to the vector registers, vector mask, real-time clock, status register, programmable clock interrupt, and the performance monitor.

## Instruction Issue

---

When an instruction issues, the scalar register receiving the data is reserved until the result is latched in the register. If an instruction in the current instruction parcel (CIP) register requires the reserved result register, that CIP instruction holds issue until the register is available. The S0 register, however, is an exception. If the S0 register is reserved as a result register and is needed as an  $S_j$  or  $S_k$  operand in a following instruction, no hold issue occurs because the S0 register has special register values as an operand.

The issue hardware also develops scalar functional unit codes. These codes select the input terms to be gated from the proper functional unit into the scalar register multiplexer.

## S Register Memory References

---

The scalar registers write or read 1 word of memory per instruction. The T registers provide intermediate storage for the scalar registers. T registers can perform memory block references, enabling a group of operands to be read from memory with one instruction. These operands are then used by the scalar registers to generate results that can be sent to the T registers and block-stored to memory. Using the T registers as buffer storage is advantageous because it takes fewer clock periods to do a block reference than to issue several individual scalar references.

The S registers also have an access path to cache memory. This provides access to common memory data without having to reference memory directly. If the requested address resides in cache, a *cache hit* is initiated and the data is read from cache instead of from common memory.

## Special Register Values

---

S0 has special register values when  $S_j$  or  $S_k$  is used as an operand. When the  $j$  field equals 0, the value sent out is 0, no matter what value is stored in S0. When the  $k$  field is 0, bit 63 is set to a 1.

## Lower/Upper Scalar Register Load

---

It is possible to load either the lower- or upper-half of a scalar register with a 32-bit quantity. The following four instructions load constants into scalar registers.

- $040i00\ mn\ Si\ exp$ : loads the quantity  $nm$  into the lower 32 bits of register  $Si$ . The upper 32 bits are cleared.
- $041i00\ mn\ Si\ exp$ : loads the one's complement of  $nm$  into the lower 32 bits of register  $Si$ . The upper 32 bits are all 1's.
- $040i20\ mn\ Si\ exp$ : loads the quantity  $nm$  into the lower 32 bits of register  $Si$ . The upper 32 bits are unchanged.
- $040i40\ mn\ Si\ exp$ : loads the quantity  $nm$  into the upper 32 bits of register  $Si$ . The lower 32 bits are unchanged.



# B AND T REGISTERS

Each CPU contains 64 (100g) B registers and 64 T registers. The B and T registers act as intermediate registers for the address and scalar registers, respectively. Each B and T register contains 64 bits.

Two BT options, BT000 and BT001, contain the B and T registers. Each option contains 32 bits of each register. BT000 contains bits 00 through 15 and 32 through 47. BT001 contains bits 16 through 31 and 48 through 63. As shown in Figure 8, the B and T registers can be loaded from the address and scalar registers, common memory, and branch control.

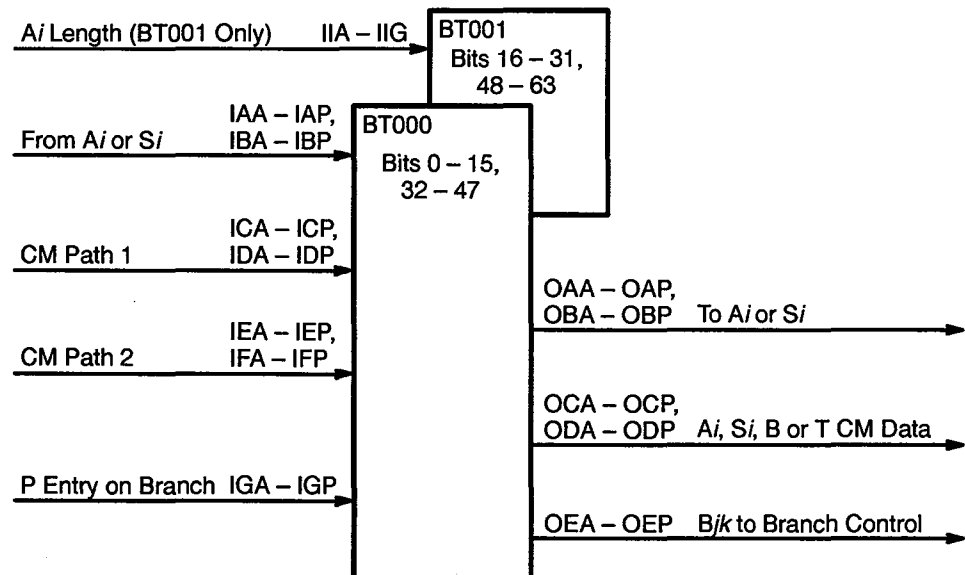


Figure 8. B and T Register Inputs and Outputs

The B and T registers are used primarily for block transfers to and from common memory. Refer to Table 2 for a list of the B and T register instructions. Refer also to Figure 9 for a B/T-register-to-memory block diagram.

Table 2. B/T Register Instructions

Instruction	CAL	Description
0050jk	J Bjk	Jump to Bjk
0051jk <sup>O</sup>	JINV Bjk	Jump to Bjk (invalidate instruction buffers)
024ijk <sup>D</sup>	Ai Bjk	Transmit (Bjk) to Ai
025ijk <sup>D</sup>	Bjk Ai	Transmit (Ai) to Bjk
034ijk <sup>D</sup>	Bjk Ai, A0	Transmit (Ai) words from common memory starting at address (A0) to B registers starting at register jk
035ijk <sup>D</sup>	,A0 Bjk,Ai	Transmit (Ai) words from B registers starting at register jk to memory starting at address (A0)
036ijk <sup>D</sup>	Tjk Ai, A0	Transmit (Ai) words from memory starting at address (A0) to T register starting at register jk
037ijk <sup>D</sup>	,A0 Tjk,Ai	Transmit (Ai) words from T registers starting at register jk to memory starting at address (A0)
074ijk	Si Tjk	Transmit (Tjk) to Si
075ijk	Tjk Si	Transmit (Si) to Tjk

O denotes a maintenance mode instruction only.

D denotes a difference between Triton mode and C90 mode.

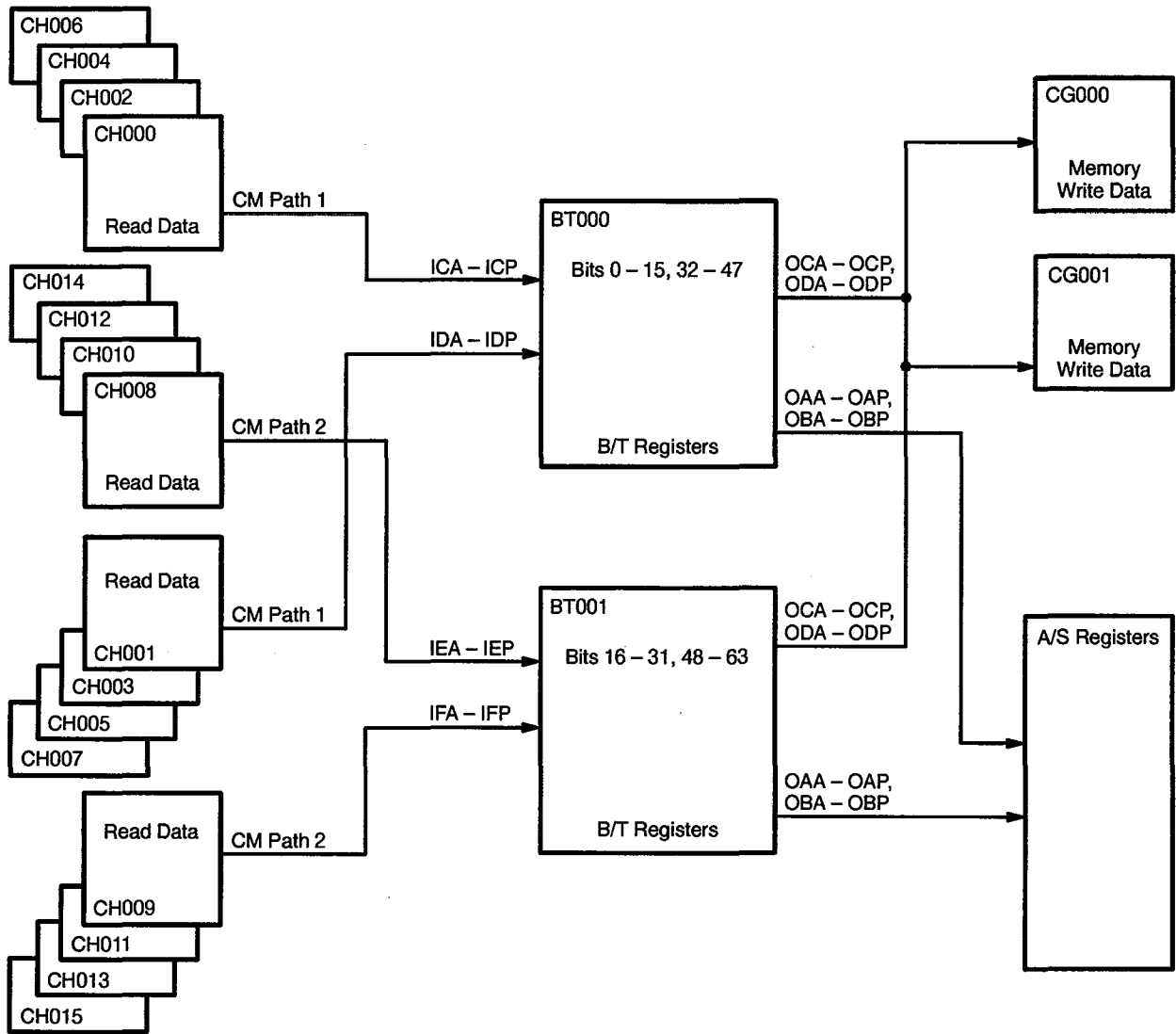


Figure 9. B/T-register-to-memory Block Diagram



## ADDRESS/SCALAR ADD

The address and scalar registers are contained on eight options: one AR option, three AS options, two AT options, and two AU options. Each option contains 8 bits of the 64-bit address registers. These options also contain the address and scalar add functional unit. Table 3 describes the instructions that use the address and scalar add functional unit.

Table 3. A/S Adder Instructions

Instruction	CAL	Description
030ijk <sup>D</sup>	$A_i A_j + A_k$	Transmit integer sum of (A <sub>j</sub> ) and (A <sub>k</sub> ) to A <sub>i</sub>
030i0k <sup>D</sup>	$A_i A_k^S$	Transmit (A <sub>k</sub> ) to A <sub>i</sub>
030ij0 <sup>D</sup>	$A_i A_j + 1^S$	Transmit integer sum of (A <sub>j</sub> ) and 1 to A <sub>i</sub>
031ijk <sup>D</sup>	$A_i A_j - A_k$	Transmit integer difference of (A <sub>j</sub> ) and (A <sub>k</sub> ) to A <sub>i</sub>
031i0k <sup>D</sup>	$A_i - A_k^S$	Transmit inverse of (A <sub>k</sub> ) to A <sub>i</sub>
031ij0 <sup>D</sup>	$A_i A_j - 1^S$	Transmit integer difference of (A <sub>j</sub> ) and 1 to A <sub>i</sub>
060ijk	$S_i S_j + S_k$	Transmit integer sum of (S <sub>j</sub> ) and (S <sub>k</sub> ) to S <sub>i</sub>
061ijk	$S_i S_j - S_k$	Transmit integer difference of (S <sub>j</sub> ) and (S <sub>k</sub> ) to S <sub>i</sub>
061i0k	$S_i - S_k$	Transmit inverse of (S <sub>k</sub> ) to S <sub>i</sub>

D denotes a difference between Triton mode and C90 mode.

S denotes a special CAL syntax.

The address add and scalar functional units perform a 64-bit add; each option performs the add function on the bits of the operands contained on that option. Carry and enable bits generated during the add are passed on to the next option, as shown in Figure 10. The 64-bit result is stored in the destination register in 4 clock periods.

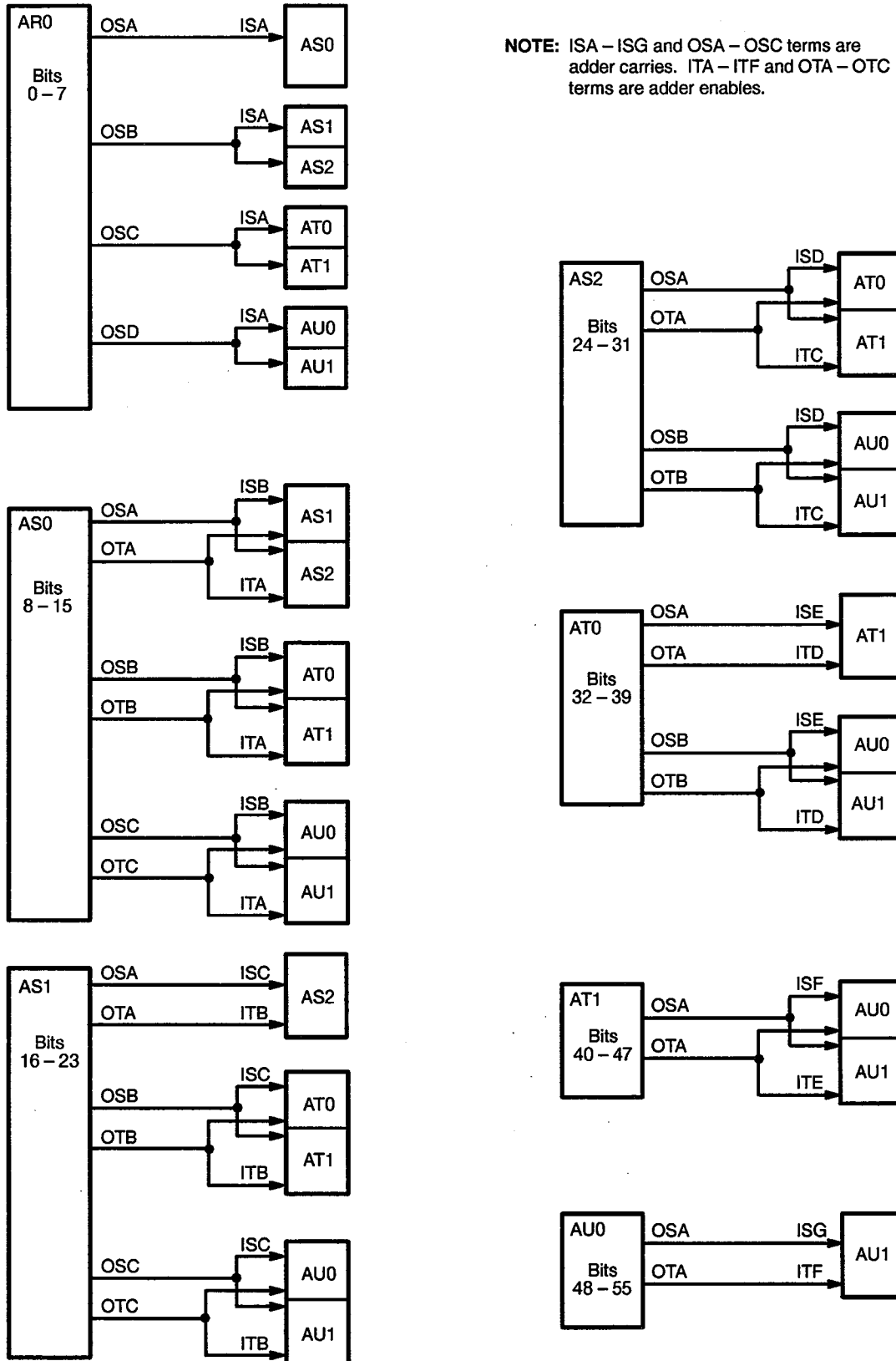


Figure 10. Carry Bit and Enable Bit Fanouts

# ADDRESS/ SCALAR LOGICAL

The scalar logical functional unit performs logical operations on the scalar registers. Logical operations include OR, AND, and XOR operations and merges.

Refer to Figure 11 for an illustration of the address/scalar registers. The scalar registers are contained on eight options: one AR option, three AS options, two AT options, and two AU options. Each option contains 8 bits of the 64-bit address registers. These options also contain the scalar logical functional unit. The operands are latched and the logical operation is completed in 1 clock period; the result is then entered into the proper destination register.

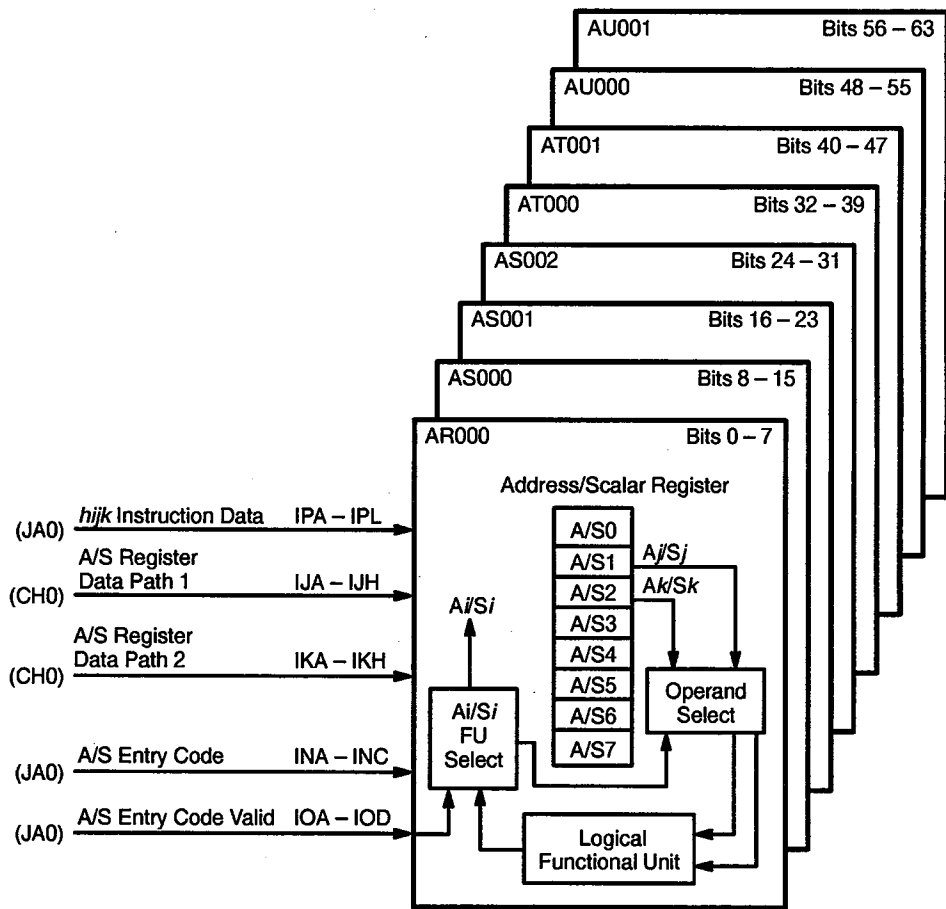


Figure 11. Address/Scalar Logical Block Diagram (Instructions 044ijk through 051ijk)

Table 4 and Table 5 list the instructions used in the address and scalar logical functional unit. The instructions listed in Table 5 must be preceded by a 005400 instruction; they are for Triton mode only.

(EIS)  
extended instruction set  
Table 4. Scalar Logical Functional Unit Instructions

Instruction	CAL	Description
044ijk	$S_i S_j \& S_k$	Logical product of (Sj) and (Sk) to Si
044i0	$S_i S_j \& SB$	Sign bit of (Sj) to Si
044i0	$S_i SB \& S_j$	Sign bit of (Sj) to Si (Sj ≠ 0)
045ijk	$S_i \# S_k \& S_j$	Logical product of (Sj) and one's complement of (Sk) to Si
045i0	$S_i \# SB \& S_j$	(Sj) with sign bit cleared to Si
046ijk	$S_i S_j \setminus S_k$	Logical difference of (Sj) and (Sk) to Si (Sj ≠ 0)
046i0	$S_i S_j \setminus SB$	Transmit (Sj) with sign bit toggled to Si
046i0	$S_i SB \setminus S_j$	Transmit (Sj) with sign bit toggled to Si (Sj ≠ 0)
047ijk	$S_i \# S_j \setminus S_k$	Logical equivalence of (Sk) and (Sj) to Si
0470k	$S_i \# S_k$	Transmit one's complement of (Sk) to Si
047i0	$S_i \# S_j \setminus SB$	Logical equivalence of (Sj) and sign bit to Si
047i0	$S_i \# SB \setminus S_j$	Logical equivalence of (Sj) and sign bit to Si (Sj ≠ 0)
04700	$S_i \# SB$	Enter one's complement of sign bit into Si
050ijk	$S_i S_j \setminus S_i \& S_k$	Logical product of (Si) and (Sk) complement ORed with logical product of (Sj) and (Sk)
050i0	$S_i S_j \setminus S_i \& SB$	Scalar merge of (Si) and sign bit of (Sj) to Si
051ijk	$S_i S_j \setminus S_k$	Logical sum of (Sj) and (Sk) to Si
0510k	$S_i S_k$	Transmit (Sk) to Si
051i0	$S_i S_j \setminus SB$	Logical sum of (Sj) and sign bit to Si (Sj ≠ 0)
05100	$S_i SB$	Enter sign bit into Si



Table 5. Address Logical Functional Unit Instructions

Instruction	CAL	Description
044ijk	$A_i A_j \& A_k$	Logical product of (A <sub>j</sub> ) and (A <sub>k</sub> ) to A <sub>i</sub>
045ijk	$A_i \# A_k \& A_j$	Logical product of (A <sub>j</sub> ) and one's complement of (A <sub>k</sub> ) to A <sub>i</sub>
046ijk	$A_i A_j \wedge A_k$	Logical difference of (A <sub>j</sub> ) and (A <sub>k</sub> ) to A <sub>i</sub> (A <sub>j</sub> ≠ 0)
047ijk	$A_i \# A_j \wedge A_k$	Logical equivalence of (A <sub>k</sub> ) and (A <sub>j</sub> ) to A <sub>i</sub>
047i0k	$A_i \# A_j$	Transmit one's complement of (A <sub>k</sub> ) to A <sub>i</sub>
050ijk	$A_i A_j \wedge A_i \& A_k$	Logical product of (A <sub>i</sub> ) and (A <sub>k</sub> ) complement ORed with logical product of (A <sub>j</sub> ) and (A <sub>k</sub> )
051ijk	$A_i A_j \vee A_k$	Logical sum of (A <sub>j</sub> ) and (A <sub>k</sub> ) to A <sub>i</sub>

## Address and Scalar Mask

Another function separate from scalar logical but included in this section, is address mask and scalar mask. Address and scalar mask functions use instructions 042ijk and 043ijk. Refer to Table 6 and Table 7 for the scalar and address mask instruction formats, respectively.

Table 6. Scalar Mask Instructions

Instruction	CAL	Description
042ijk	$S_i \lt \text{exp}$	Form ones mask in S <sub>i</sub> exp bits from the right; jk field = 100 - exp
042i77	$S_i 1$	Enter 1 into S <sub>i</sub>
042i00	$S_i -1$	Enter -1 into S <sub>i</sub> ; (S <sub>i</sub> = 177777 177777 177777 177777)
043ijk	$S_i \gt \text{exp}$	Form ones mask in S <sub>i</sub> exp bits from the left: jk field = exp
043ijk	$S_i \# \lt \text{exp}$	Form zeroes mask in S <sub>i</sub> exp bits from the right: jk field gets 100 <sub>8</sub> = exp
043i00	$S_i 0$	Clear S <sub>i</sub>

Table 7. Address Mask Instructions

Instruction	CAL	Description
042ijk	$A_i < exp$	Form ones mask in $A_i$ $exp$ bits from the right; $jk$ field = $100 - exp$
042i77	$A_i 1$	Enter 1 into $A_i$
042i00	$A_i -1$	Enter -1 into $A_i$ ; ( $A_i = 177777 177777 177777 177777$ )
043ijk	$A_i > exp$	Form ones mask in $A_i$ $exp$ bits from the left: $jk$ field = $exp$
043ijk	$A_i \# < exp$	Form zeroes mask in $A_i$ $exp$ bits from the right: $jk$ field gets $100_8 = exp$
043i00	$A_i 0$	Clear $A_i$

The address/scalar mask functional unit is located on the SS options. When the 042ijk or 043ijk instruction issues the *jk* field, it is sent from the BT0 option. The *jk* field determines how many 1 bits are set, and the *h* field bit 0 determines whether the one's should be on the left or the right. Figure 12 is a block diagram of the scalar mask functional unit.

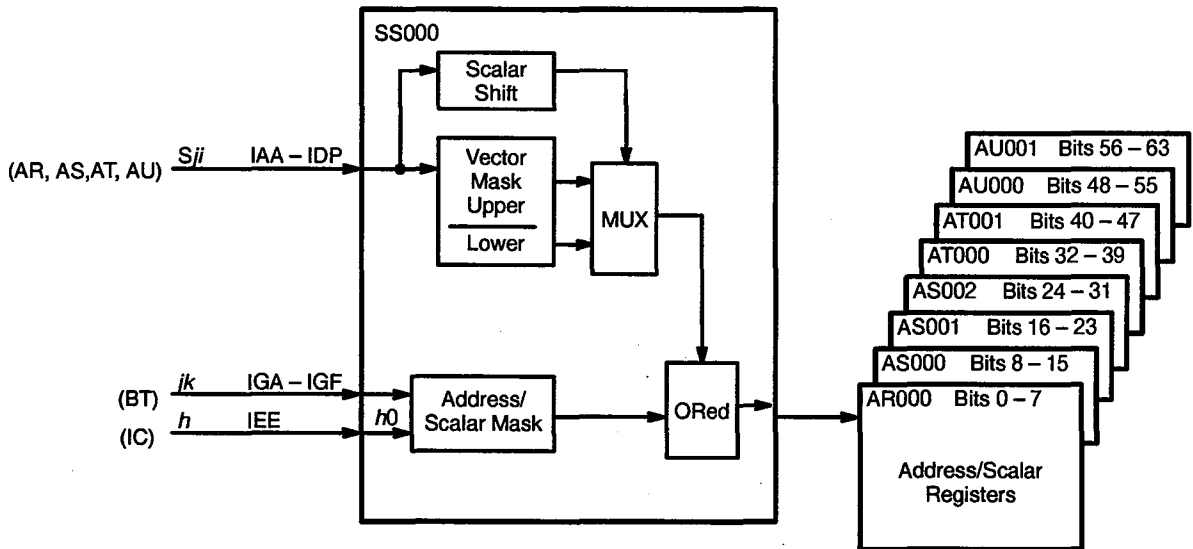


Figure 12. Scalar Mask Block Diagram

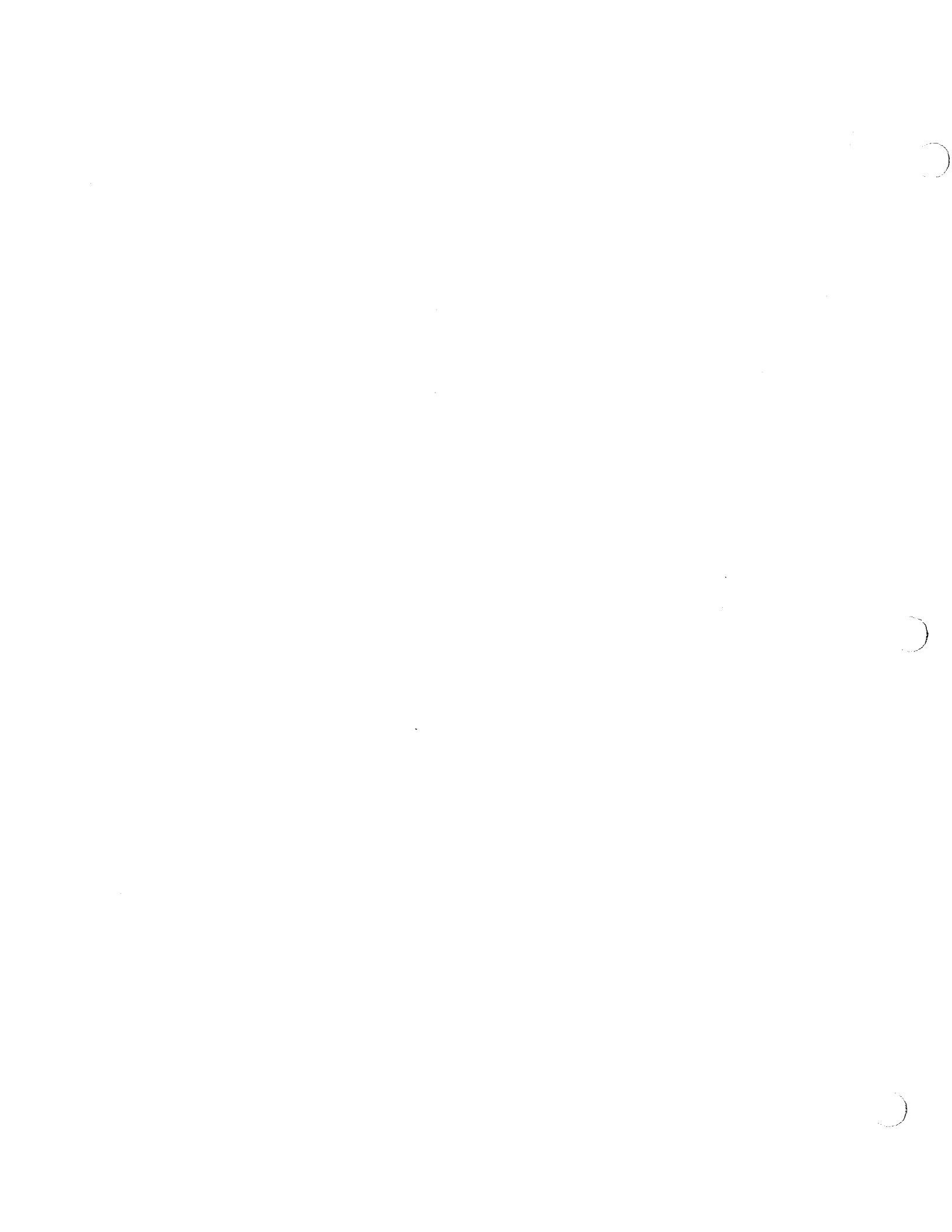
## Transmit *nm* to *Si*, *Si* Upper, *Si* Lower

---

Constant data can be transmitted to an S register by four different instructions. Refer to Table 8 for a list of these instructions.

Table 8. Transmit *nm* to *Si* Instructions

Instruction	CAL	Description
040/00 <i>nm</i>	<i>Si exp</i>	Transmit expression = <i>nm</i> to <i>Si</i> , bits 0 through 31 (bits 32 through 63 = 0)
040/20 <i>nm</i>	<i>Si Si.exp</i>	Transmit expression = <i>nm</i> to <i>Si</i> , bits 0 through 31 (bits 32 through 63 unchanged) ( <i>j</i> 2 = 0)
040/40 <i>nm</i>	<i>Si exp.Si</i>	Transmit expression = <i>nm</i> to <i>Si</i> , bits 32 through 63 (bits 0 through 31 unchanged) ( <i>j</i> 2 = 1)
041/00 <i>nm</i>	<i>Si exp</i>	Transmit expression = one's complement of <i>nm</i> to <i>Si</i> , bits 0 through 31 ( <i>Si</i> bits 32 through 63 = 1)



## ADDRESS/SCALAR POP/PARITY AND LEADING ZERO

The address/scalar population count functional unit counts the number of 1 bits in the scalar (S) register or address (A) register of the  $k$  field of instruction  $026ijk$  ( $k = 0$  or  $1$  for S registers, and  $k = 2$  or  $3$  for A registers). The maximum count could be  $100_8$  or  $64_{10}$  for the corresponding number of 1 bits set in the A or S register, and the smallest count could be 0 when no bits are set in the A or S register.

The  $k$  field of the instruction determines whether or not the entire population count is recorded in  $A_i$ . If it is a  $026ij0/2$  instruction, all 7 bits of the final population count are sent to the A register. When a  $026ij1/3$  instruction is issued, the entire S or A register is counted for the number of 1 bits set, but then only bit 0 of the count is sent to the A register. If bit 0 of the count equals 0, then the count has even parity, indicating an even number of bits set. If bit 0 of the count equals 1, then the count has odd parity.

Starting from bit position 63, the address/scalar leading zero count functional unit counts the number of 0's preceding the first bit set to a 1 in a specified address or scalar register. The number of leading 0's is then transferred to the lower 7 bits of an  $A_i$  register. To use the address/scalar leading zero count functional unit, a  $027ij0$  instruction is issued when  $S_j$  is the operand and  $A_i$  is the result register. The  $027ij1$  is issued when  $A_j$  is the operand and  $A_i$  is the result register.

The SS option performs scalar pop/parity and leading zero functions. Population count/parity and leading zero functions are performed on either a scalar or an address register operand, with the result sent to an address register. Table 9 describes the instructions that use the pop/parity and leading zero functional unit, and Figure 13 illustrates the A/S population/parity/leading zero count.

Table 9. Scalar Pop Count/Parity and Leading Zero Count Instructions

Instruction	CAL	Description
026ij0 <sup>D</sup>	$A_i PS_j$	Transmit population count of (S) to $A_i$
026ij1 <sup>D</sup>	$A_i QS_j$	Transmit population count parity of (S) to $A_i$
026ij2 <sup>ND</sup>	$A_i PA_j$	Transmit population count of (A) to $A_i$
026ij3 <sup>ND</sup>	$A_i QA_j$	Transmit population count parity of (A) to $A_i$
027ij0	$A_i ZS_j$	Transmit leading zero count of (S) to $A_i$
027ij1 <sup>NT</sup>	$A_i ZA_j$	Transmit leading zero count of (A) to $A_i$

D denotes a difference between Triton mode and C90 mode.

N denotes new instruction (not available on CRAY C90 series systems).

T denotes Triton mode only.

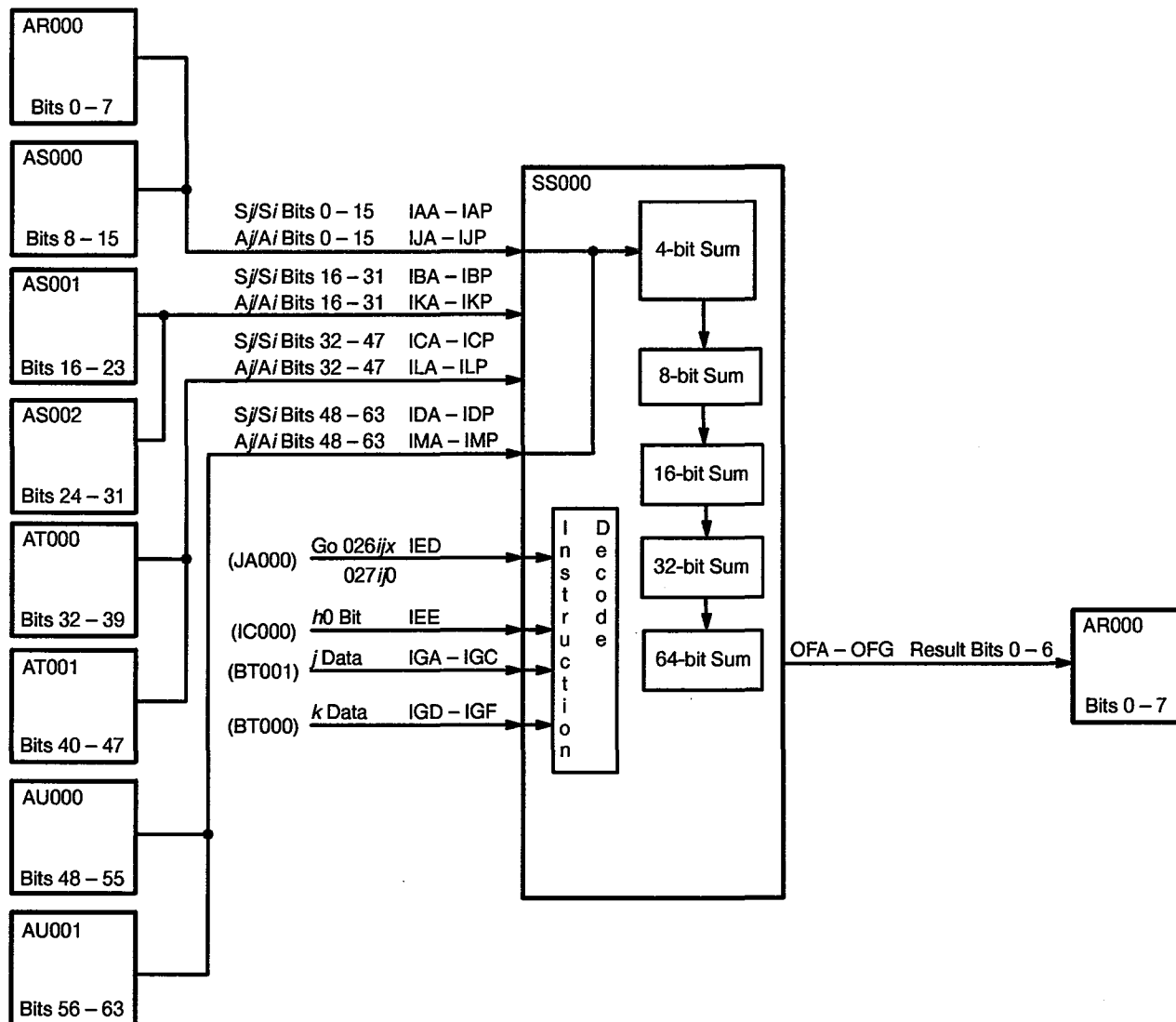
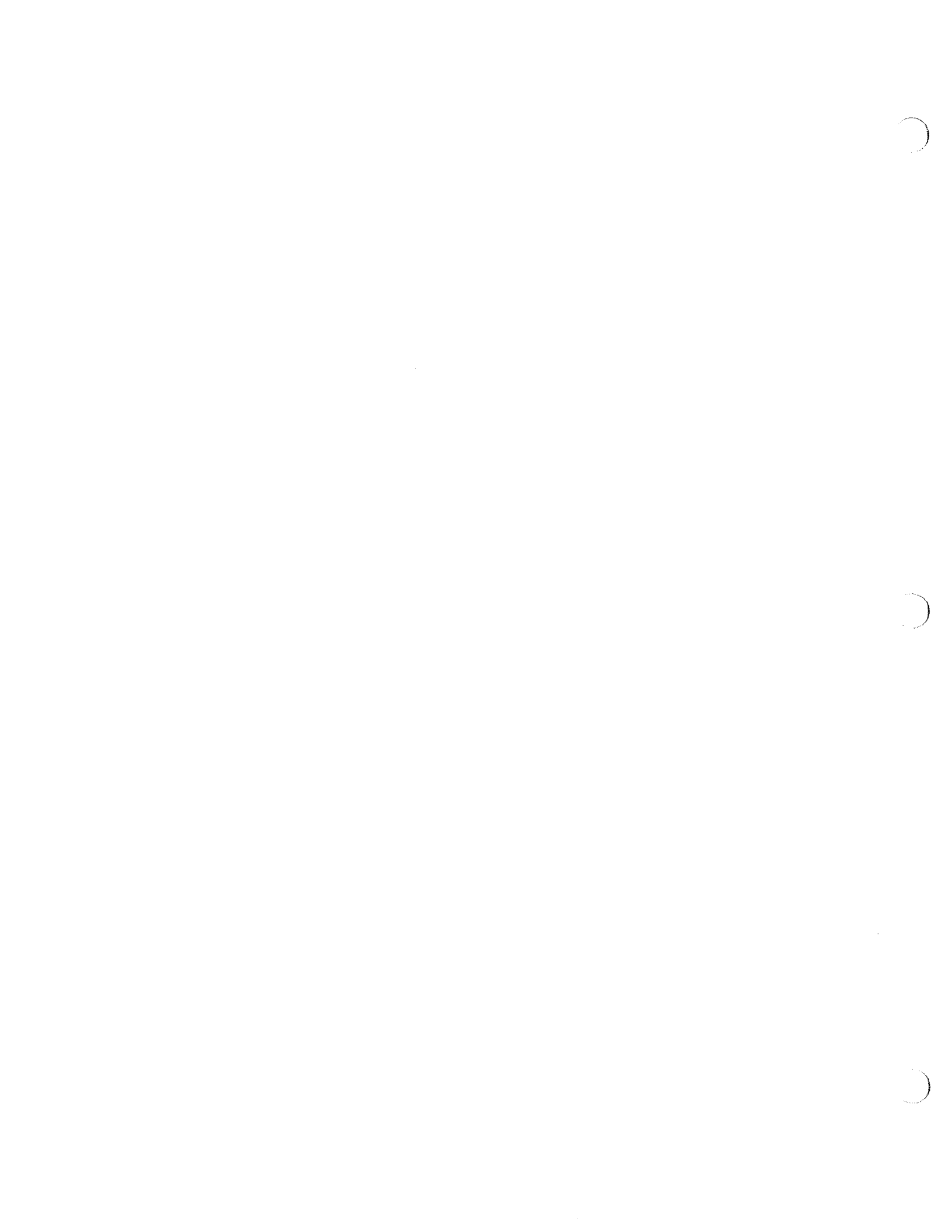


Figure 13. A/S Population/Parity/Leading Zero Count





# ADDRESS REGISTER SHIFT

The address register shift function is performed on the SS option (refer to Figure 14 for a block diagram of address register shift). This functional unit performs both left and right single-register shifts and left and right double-register (also referred to as *long*) shifts. All shifts are end-off with zero fill. For example, if data is shifted more than  $64_{10}$  places in a single shift, or more than  $128_{10}$  places in a double-register shift, the data is shifted off the register. The data is then lost, and 0's are moved into the register.

The shift unit performs only left shifts. The shift count for a right shift must be in the two's complement form; the unit then performs a left shift. Refer to Table 10 for a list of the address register shift instructions.

**NOTE:** To issue A-register-shift instructions, a 005400 (EIS) instruction must precede the shift instruction. If an A-register-shift instruction is issued in C90 mode, the results are undefined.

Table 10. Address Register Shift Instructions

Instruction	CAL	Description
052ijk	A0 A<exp	Shift (A <sub>i</sub> ) left exp = jk places to A0
053ijk	A0 A>exp	Shift (A <sub>i</sub> ) right exp = $100_8 - jk$ places to A0
054ijk	Ai A<exp	Shift (A <sub>i</sub> ) left exp = jk places to A <sub>i</sub>
055ijk	Ai A>exp	Shift (A <sub>i</sub> ) right exp = $100_8 - jk$ places to A <sub>i</sub>
056ijk	Ai Ai, Aj<Ak	Shift (A <sub>i</sub> ) and (A <sub>j</sub> ) left (A <sub>k</sub> ) places to A <sub>i</sub>
056i0	Ai Ai, Aj<1	Shift (A <sub>i</sub> ) and (A <sub>j</sub> ) left one place to A <sub>i</sub>
0560k	Ai A<Ak	Shift (A <sub>i</sub> ) left (A <sub>k</sub> ) places to A <sub>i</sub>
057ijk	Ai Aj, A>Ak	Shift (A <sub>j</sub> ) and (A <sub>i</sub> ) right (A <sub>k</sub> ) places to A <sub>i</sub>
057i0	Ai Aj, A>1	Shift (A <sub>j</sub> ) and (A <sub>i</sub> ) right one place to A <sub>i</sub>
0560k	Ai A>Ak	Shift (A <sub>i</sub> ) right (A <sub>k</sub> ) places to A <sub>i</sub>

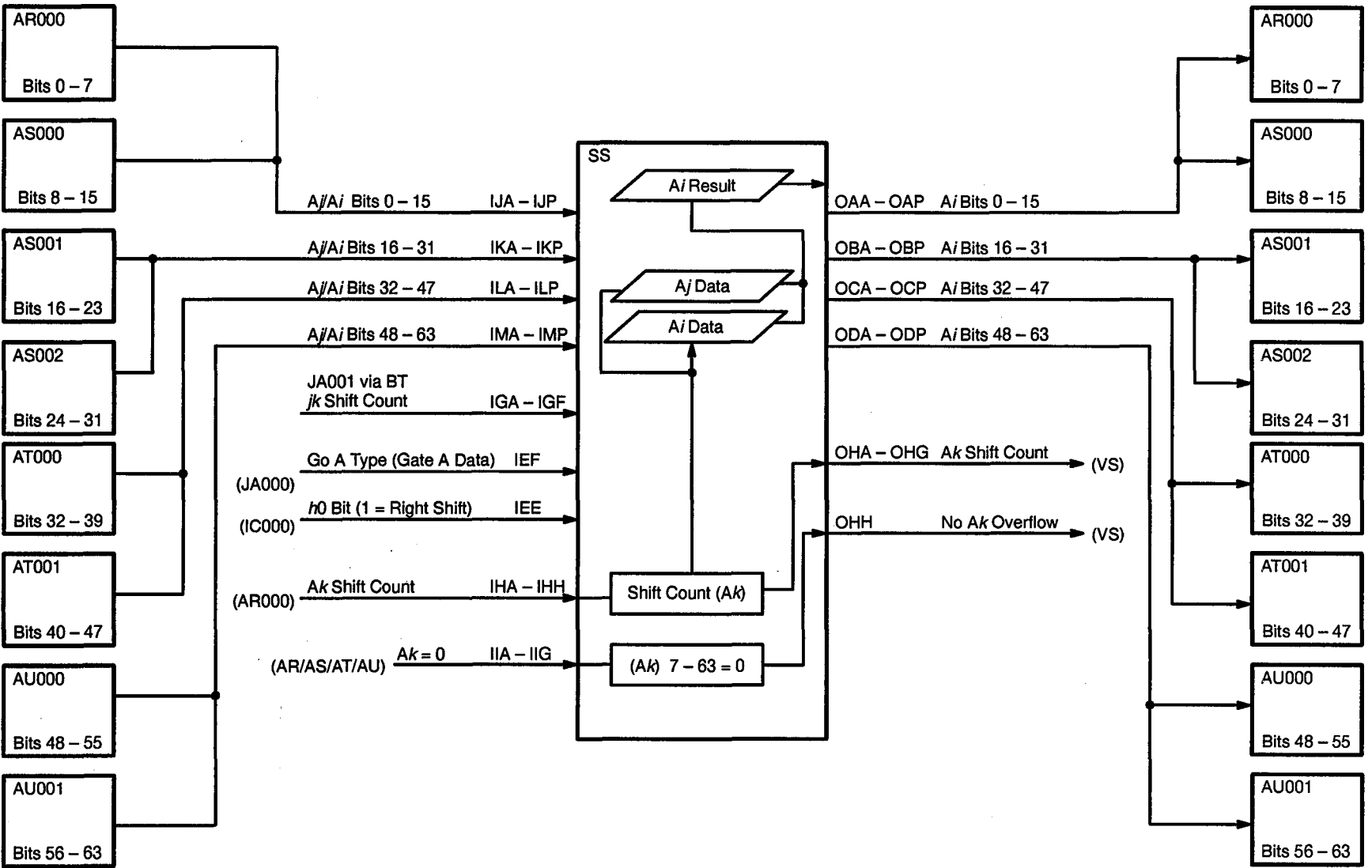


Figure 14. Address Register Shift

## Address Register Single Shift

---

The address register single-shift instructions are  $052ijk$  through  $055ijk$ . The first two instructions perform left single shifts ( $052ijk$ ) and right single shifts ( $053ijk$ ) on the content of the  $A_i$  register and always store the result in  $A_0$ . The shift count is obtained from the  $jk$  field of the instruction. The value placed in the  $jk$  field for the single-shift instructions depends on whether it is a left or right shift. For a single left shift, the value in the  $jk$  field is the number of octal places desired to shift  $A_i$ . This allows a shift left of 0 to  $77_8$  places. For a right shift, the  $jk$  field is equal to the two's complement of the actual number of places desired to shift right. If a shift of  $24_8$  places were required,  $54$  would be entered in the  $jk$  field (two's complement of  $24$  is  $54$ ).

When instructions are written in machine code, this operation must be done by the person writing the code. However, when instructions are written in CAL, this is done by the assembler. In the CAL instruction, you would simply enter the shift count. This allows a shift right of 1 to  $100_8$  places. Because the two's complement of the shift count is used for a single shift, a shift right 0 places is not possible.

The  $054ijk$  and  $055ijk$  instructions perform single shifts left or right on the contents of  $A_i$ . However, these instructions store the result of the shift back in  $A_i$ . These shifts overwrite the original contents of  $S_i$  with the new results from the shifter.

## Address Register Double Shift

---

Double shifts work similar to single shifts and are end-off with zero fill. The difference is that a double shift concatenates two S registers, forming a 128-bit register. The arrangement of the two registers is determined by the shift direction.

Double shifts always shift data into  $S_i$ . The two instructions associated with double shifts are  $056ijk$  (left double shift) and  $057ijk$  (right double shift). The double shifts use the  $i$  and  $j$  fields to specify the two operand registers; the  $i$  field also specifies the result register. The  $k$  field of the instructions specifies the A register used for the shift count.

Because a double shift uses a 128-bit operand and shifts are end-off with zero fill, a shift equal to or greater than  $128_{10}$  ( $200_8$ ) produces a result of zero. The A register bits 0 through 6 are used as a shift count, providing a shift of 0 to  $177_8$ . Bit 7 is checked, and if this bit is set to a 1, it causes the double shift result to equal zero. For right double shifts, the shift count does not need to be entered into the A register in two's complement form; the hardware performs this function.

## Address Register Shift Count Description

---

The AR option sends 7 bits of shift count to the SS option. For both single and double shifts, the breakdown of the shift count is the same, except that the double shift has 1 extra bit (bit 6). Refer to Figure 15 for a breakdown of the shift count.

Double Shift Only								
6	5	4	3	2	1	0	Bit Position	
64	32	16	8	4	2	1	Shift Value	

Figure 15. Shift Count Breakdown

Each bit position of the shift count represents a shift value, and the sum of the shift value for each bit set in the shift count equals the total number of places shifted.

**NOTE:** The shift value is shown as a decimal value; all references to shift counts in the documentation refer to a decimal count.

If the *jk* field of a left single shift equals  $27_8$  and bits 4, 2, 1, and 0 are set, the shift values would be 16, 4, 2, and 1, respectively. The sum of the shift values would be 23 ( $16 + 4 + 2 + 1$ ); therefore, the instruction would shift left  $23_{10}$  places.

The actual hardware that performs the shifts is the same for both left and right shifts. However, the hardware performs only left shifts. Right shifts are accomplished by the way in which data is entered into the shifter, hence the use of two's complement for right shifts.

## Address Register Left Single Shift

Figure 16 is an illustration of how a left single shift is performed for a 054220 instruction. ( $A_i A_{i < exp}$ ), shift A2 left  $jk$  places ( $20g$ ) with data bit 10 set.

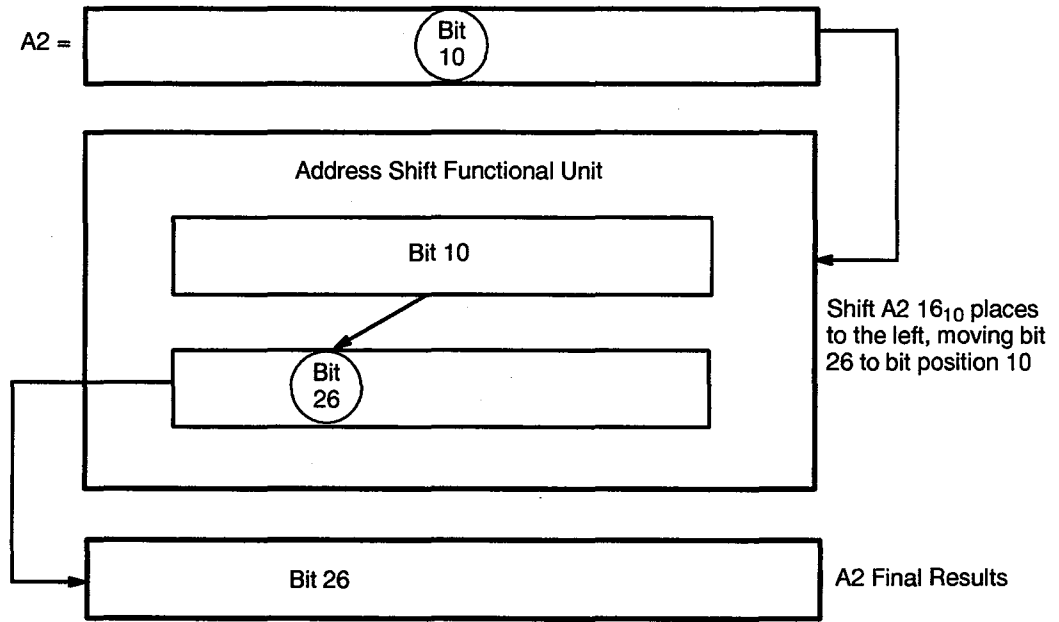


Figure 16. Address Register Left Single Shift

## Address Register Right Single Shift

Figure 17 is an illustration of how a right single shift is performed using left shifts and a two's complement shift count. This example uses a 055254 instruction ( $A_i > A_i \text{ exp}$ ) that shifts  $A_i$  right  $\text{exp} = 100 - jk$  places to  $A_i$ . In this example, data bit 45 shifts to the right  $24_8$  ( $20_{10}$ ) places. Notice that the  $jk$  field of the instruction 055254 contains  $54_8$ , which is the two's complement of  $24_8$ , causing A2 to be shifted to the left  $54_8$  places to set bit 25 of the result.

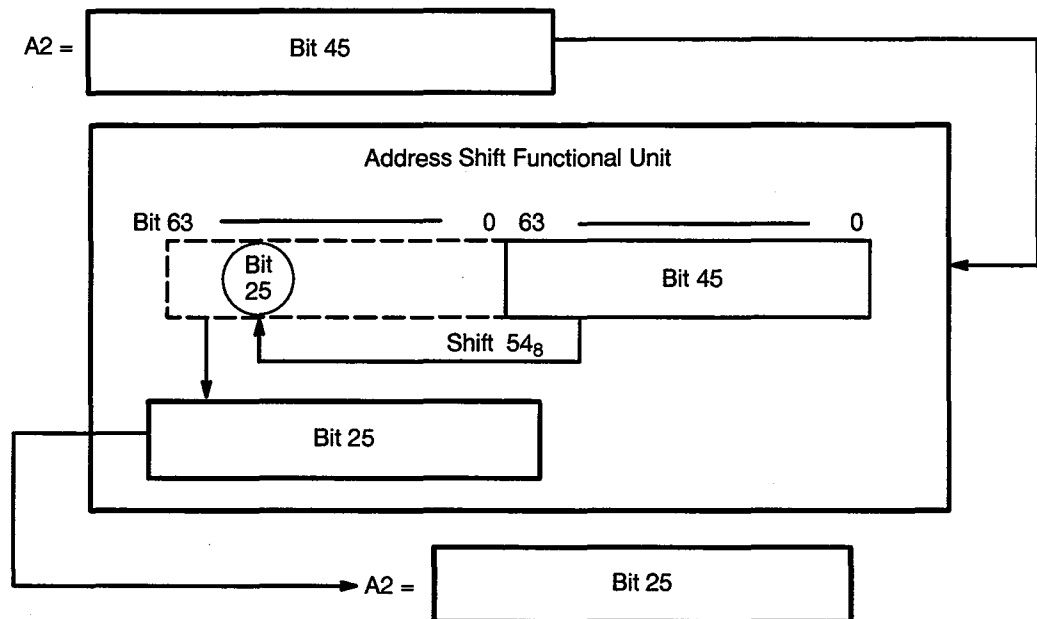


Figure 17. Address Register Right Single Shift

**NOTE:** On a right shift, it is the programmer's responsibility to perform the two's complement of the shift count and supply that value to the functional unit.

## Address Register Left Double Shift

Double shifts are the same as single shifts except that they concatenate two 64-bit registers to form a value. Figure 18 is an illustration of a left double shift using a 056123 instruction ( $A_i A_1, A_j < A_k$ ). In this example, we shift ( $A_i$ ) and ( $A_j$ ) left ( $A_k$ ) places to  $A_i$ , with  $A_3 = 40_8 (32_{10})$ ,  $A_1$  having bit 30 set, and  $S_2$  having bit 10 set. When a left double shift occurs, the content of  $A_j$  is moved into  $A_i$ , and the two registers are positioned as shown with  $A_i$  ahead of  $A_j$ .

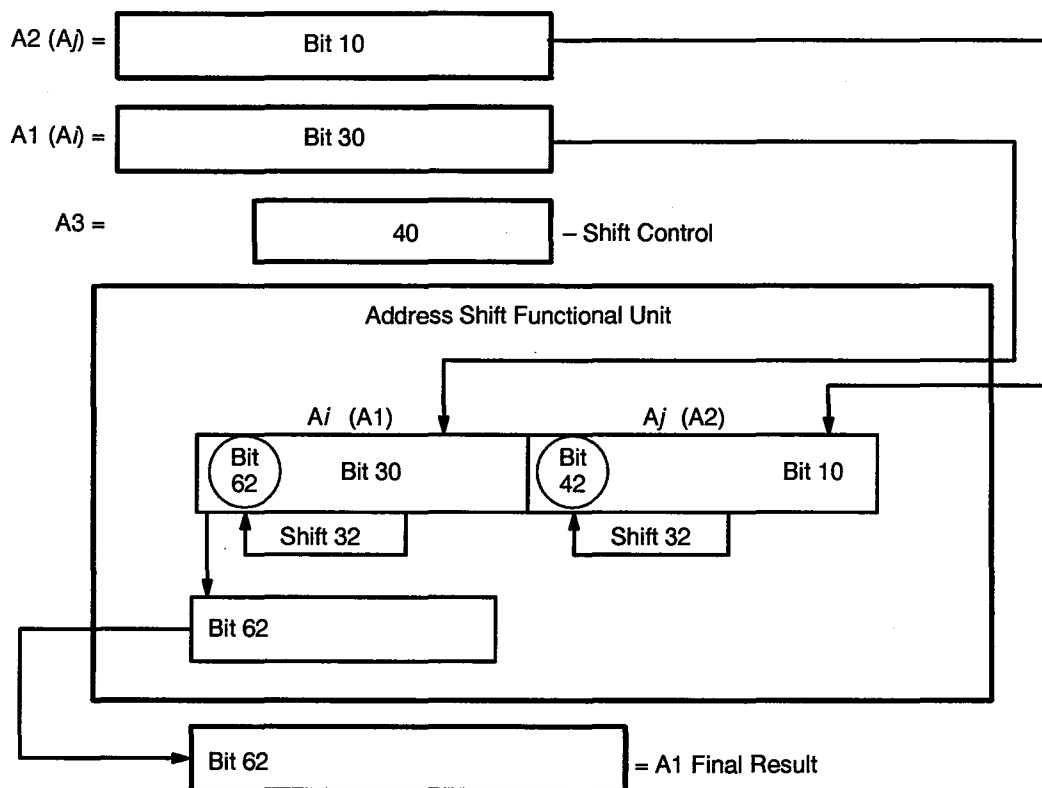


Figure 18. Address Register Left Double Shift

Shifting  $A_i$  and  $A_j$  to the left 32 places puts bit 30 of  $A_1$  at bit position 62 and bit 10 of  $A_2$  at bit position 41. Because bit 41 of  $A_2$  did not make it to the result register  $A_1$ , it is lost. The result bit (bit 62) is then sent to the  $A_i (A_1)$  register. The  $A_j (A_2)$  register remains changed.

## Address Register Right Double Shift

To perform an address register right double shift, a 057ijk [( $A_i A_j$ ,  $A_i > A_k$ ), shift ( $A_j$ ) and ( $A_i$ ) right ( $A_k$ ) places to  $A_i$ ] instruction is used. Figure 19 illustrates a 057123 instruction with the indicated parameters.

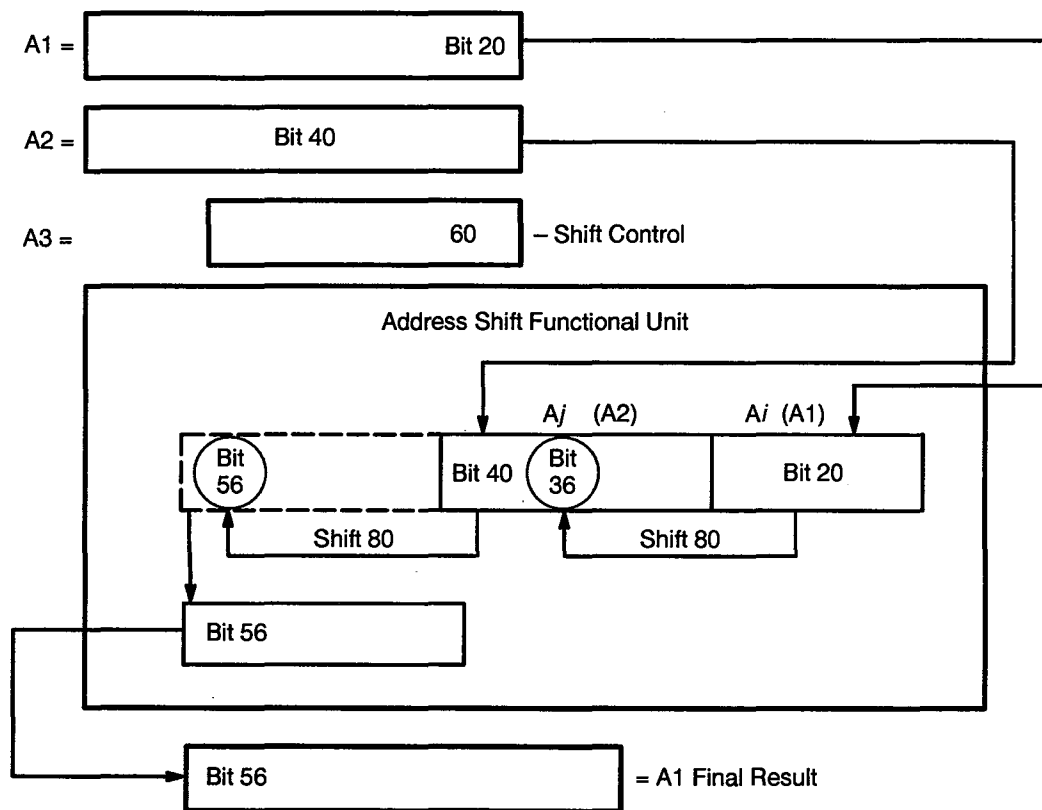


Figure 19. Address Register Right Double Shift

To right shift  $A_j$  and  $A_i$  using left shifts, the two's complement is first performed on  $A_3$ , which currently equals  $60_8$  ( $48_{10}$ ). Because the two's complement is  $120_8$  (or  $1010000_2$  or  $80_{10}$ ), the required shift can be accomplished through successive shifts of  $64_{10}$  and  $16_{10}$  for a total shift of  $80_{10}$  places. A left shift of  $80_{10}$  would move bit 40 of  $A_2$  to bit position 56 inside the dotted box and bit 20 of  $A_1$  to bit position 36 of  $A_2$ . Because bit 36 did not make it into the result register (indicated by the dotted box), it is lost, and bit 56 is sent to the final result.



## Left Single-shift Instruction

Refer to Figure 20 when reading the two following examples of the address register left single-shift instruction.

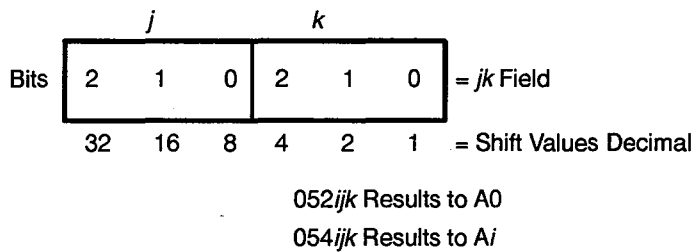


Figure 20. Example of an A Register Left Single-shift Instruction

**Example 1:** Write the instruction to shift A2 left 20<sub>10</sub> places, putting the results into A0.

- Steps:
1. 052*ijk* – left shift instruction result goes to A0
  2. *jk* field – shift count 20<sub>10</sub> = 24<sub>8</sub> = *jk* field
  3. 052224 – final instruction

**Example 2:** Write the instruction to shift A4 left 35<sub>10</sub> places, putting the results into A4.

- Steps:
1. 054*ijk* – left shift instruction result goes to A*i*
  2. *jk* field – shift count 35<sub>10</sub> = 43<sub>8</sub>
  3. 054443 – final instruction

## Right Single-shift Instruction

---

The right single-shift count is the  $jk$  field of the instruction, which must either be in the two's complement form or  $100_8$  minus the number of places to right shift. The following two examples show an address register right single-shift instruction.

- $053ijk$  results to  $A0$
- $055ijk$  results to  $Ai$

Example 1: Write the instruction to shift  $A5$  right  $10_{10}$  places, putting the results into  $A0$ .

- Steps: 1.  $053ijk$  – right shift instruction results to  $A0$
2.  $jk$  field – shift count in two's complement equals  $66_8$

$$10_{10} = 12_8 = 001010$$

$$\text{two's complement} = 110101$$

$$\begin{array}{r} + 1 \\ \hline 110110 = 66_8 \end{array}$$

3.  $053566$  – final instruction

Example 2: Write the instruction to shift  $A7$  right  $28_{10}$  places.

- Steps: 1.  $055ijk$  right shift instruction results to  $Ai$
2.  $jk$  field – shift count in two's complement equals

$$28_{10} = 34_8 = 011100$$

$$\text{two's complement} = 100011$$

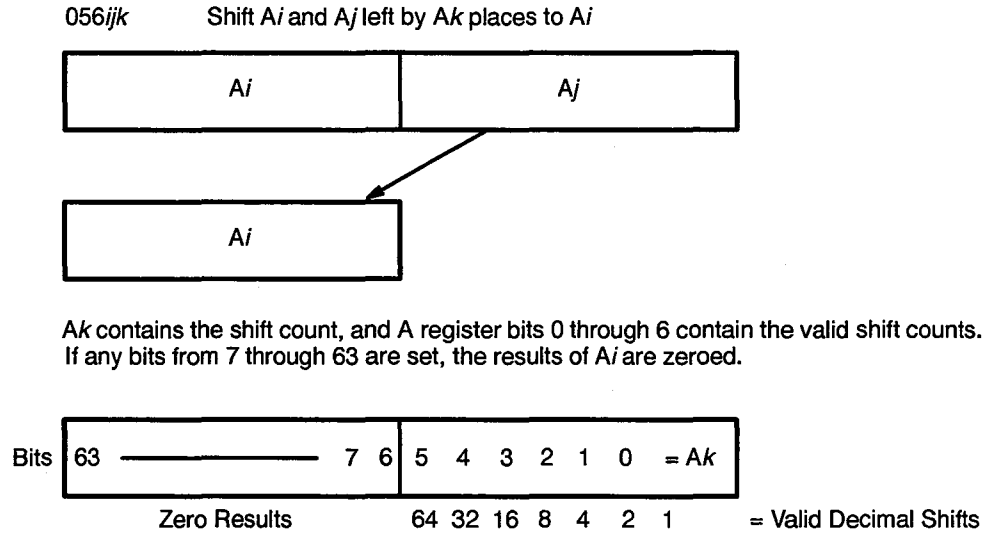
$$\begin{array}{r} + 1 \\ \hline 100100 = 44_8 \end{array}$$

$$\text{or } 100_8 - 34_8 = 44_8$$

3.  $055744$  – final instruction

## Left Double-shift Instruction

Refer to Figure 21 when reading the following example of an address register left double-shift instruction.



On a left double shift, the contents of  $A_j$  are always shifted into  $A_i$ . This shift is done inside the address shift functional unit.

Figure 21. Example of an Address Register Left Double-shift Instruction

**Example 1:** Write the instruction to left double shift  $A_2$  and  $A_3$   $64_{10}$  places, putting the results into  $A_2$ .

056234 – final instruction, where  $A_4 = 100_8$

**NOTE:** A circular left shift can be effected by issuing a 056 instruction with  $i = j$  and  $(A_k) \leq 64$ .

## Right Double-shift Instruction

Refer to Figure 22 when reading the following example of a scalar right double-shift instruction.

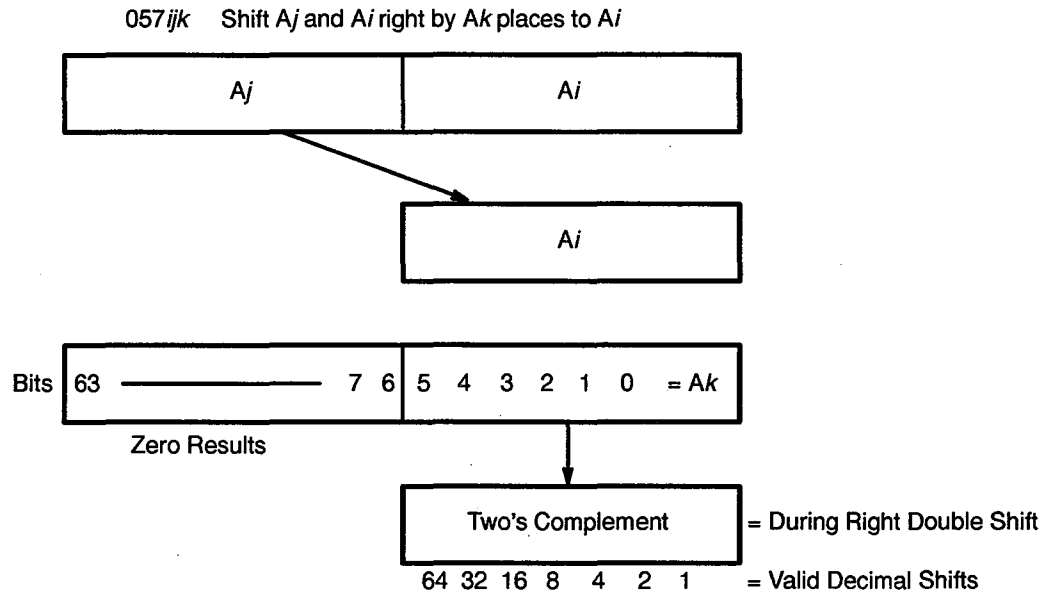


Figure 22. Example of an Address Register Right Double-shift Instruction

$A_k$  contains the shift count, and address (A) register bits 0 through 6 contain the valid shift counts. If any bits from 7 through 63 are set, the results of  $A_i$  are zeroed. Also, the hardware generates the two's complement of the shift count  $A_k$  register bits 0 through 6 on a right double shift.

On a right double shift, the contents of  $A_j$  are always shifted into  $A_i$ . This operation and the two's complement of the shift count are done inside the address shift functional unit.

**Example 1:** Write the instruction to right double shift  $A_4$  and  $A_5$  32<sub>10</sub> places, with the results going into  $A_4$ .

057454 – final instruction, where  $A_4 = 40_8$   
hardware generates a shift count of 140<sub>8</sub> inside  
the functional unit.

**NOTE:** A circular right shift can be effected by issuing a 057 instruction with  $i = j$  and  $(A_k) \leq 64$ .

# SCALAR SHIFT

The scalar shift function is performed on the SS option (refer to Figure 23 for a block diagram of a scalar shift). This functional unit performs both left and right single-register shifts, and left and right double-register (also referred to as *long*) shifts. All shifts are end-off with zero fill. For example, if data is shifted more than  $64_{10}$  places in a single shift, or more than  $128_{10}$  places in a double-register shift, the data is shifted off the register. The data is then lost, and the register is filled with 0's.

The shift unit performs only left shifts. The shift count for a right shift has to be in the two's complement form; the unit then performs a left shift. Refer to Table 11 for a list of the scalar shift instructions.

Table 11. Scalar Shift Instructions

Instruction	CAL	Description
052ijk	S0 S<exp	Shift (S <sub>i</sub> ) left exp = jk places to S0
053ijk	S0 S>exp	Shift (S <sub>i</sub> ) right exp = $100_8 - jk$ places to S0
054ijk	S <sub>i</sub> S<exp	Shift (S <sub>i</sub> ) left exp = jk places to S <sub>i</sub>
055ijk	S <sub>i</sub> S>exp	Shift (S <sub>i</sub> ) right exp = $100_8 - jk$ places to S <sub>i</sub>
056ijk	S1 S <sub>i</sub> , S <sub>j</sub> <Ak	Shift (S <sub>i</sub> ) and (S <sub>j</sub> ) left (Ak) places to S <sub>i</sub>
056ij0 <sup>†</sup>	S1 S <sub>i</sub> , S <sub>j</sub> <1	Shift (S <sub>i</sub> ) and (S <sub>j</sub> ) left 1 place to S <sub>i</sub>
056i0k <sup>‡</sup>	S1 S <sub>i</sub> <Ak	Shift (S <sub>i</sub> ) left (Ak) places to S <sub>i</sub>
057ijk	S <sub>i</sub> S <sub>j</sub> , S <sub>i</sub> >Ak	Shift (S <sub>j</sub> ) and (S <sub>i</sub> ) right (Ak) places to S <sub>i</sub>
057ij0 <sup>†</sup>	S1 S <sub>j</sub> , S <sub>i</sub> >1	Shift (S <sub>j</sub> ) and (S <sub>i</sub> ) right 1 place to S <sub>i</sub>
057i0k <sup>‡</sup>	S1 S <sub>i</sub> >Ak	Shift (S <sub>i</sub> ) right (Ak) places to S <sub>i</sub>

<sup>†</sup> If j = 0, then (S<sub>j</sub>) = 0

<sup>‡</sup> If k = 0, then (Ak) = 1

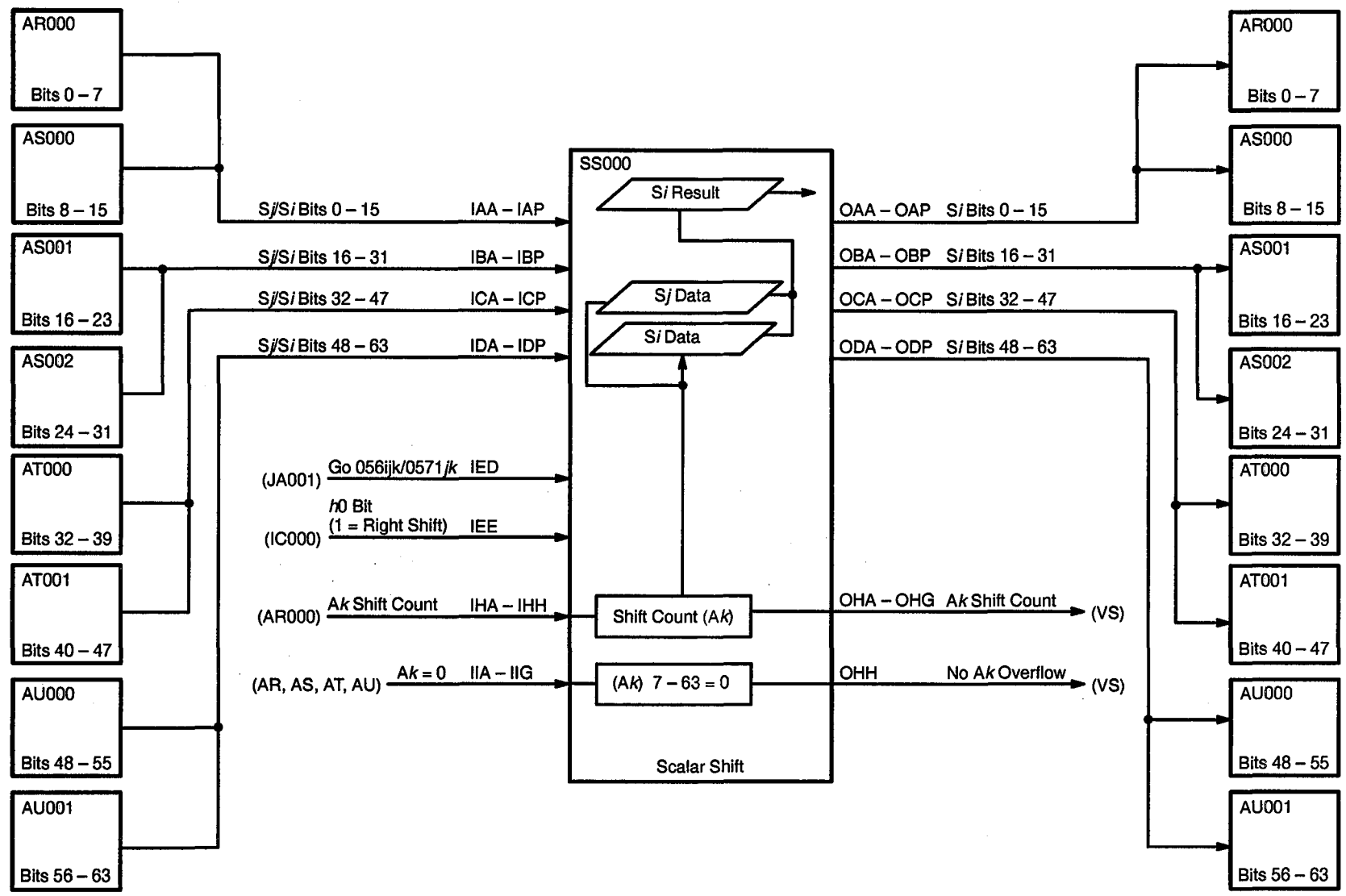


Figure 23. Scalar Shift

## Scalar Single Shift

---

The scalar single-shift instructions are  $052ijk$  through  $055ijk$ . The first two instructions perform single shifts left ( $052ijk$ ) and right ( $053ijk$ ) on the contents of the  $S_i$  register and always store the result in  $S_0$ . The shift count is obtained from the  $jk$  field of the instruction. The value placed in the  $jk$  field for the single-shift instructions depends on whether it is a left or right shift. For a single left shift, the value in the  $jk$  field is the number of octal places desired to shift  $S_i$ . This allows a shift left of 0 to  $77_8$  places. For a right shift, the  $jk$  field is equal to the two's complement of the actual number of places desired to shift right. If a shift of  $24_8$  places were required,  $54$  would be entered in the  $jk$  field (two's complement of  $24$  is  $54$ ).

When instructions are written in machine code, this operation must be done by the person writing the code. However, when instructions are written in CAL, this operation is done by the assembler. In the CAL instruction, you would simply enter the shift count. This allows a right shift of 1 to  $100_8$  places. Because the two's complement of the shift count is used for a single shift, a shift right of 0 places is not possible.

The  $054ijk$  and  $055ijk$  instructions perform single shifts left or right on the contents of  $S_i$ . However, these instructions store the result of the shift back in  $S_i$ . These shifts overwrite the original contents of  $S_i$  with the new results from the shifter.

## Scalar Double Shift

---

Double shifts work similar to single shifts; all shifts are end-off with zero fill. The difference is that a double shift concatenates two S registers, forming a 128-bit register. The arrangement of the two registers is determined by the shift direction.

Double shifts always shift data into  $S_i$ . The two instructions associated with double shifts are  $056ijk$  (double left shift) and  $057ijk$  (double right shift). The double shifts use the  $i$  and  $j$  fields to specify the two operand registers; the  $i$  field also specifies the result register. The  $k$  field of the instructions specifies the A register used for the shift count.

Because a double shift uses a 128-bit operand and shifts are end-off with zero fill, a shift equal to or greater than  $128_{10}$  ( $200_8$ ) produces a result of zero. The A register bits 0 through 6 are used as a shift count, providing a shift of 0 to  $177_8$ . For right double shifts, the shift count does not need to be entered into the A register in two's complement; the hardware performs this function.

## Scalar Shift Count Description

---

The AR000 option sends the shift count to the SS option. All eight A-series options check the value of the 64-bit A register to discover whether any bits above bit 6 have been set. If any bits have been set, the result is lost due to overshift. If each A-series option reports that its bits are zero, a signal called  $Ak = 0$  is sent to the SS option and the shift count is valid.

The AR option sends 7 bits of shift count to the SS option. For both single and double shifts, the breakdown of the shift count is the same, except for the fact that the double shift has 1 extra bit (bit 6). Refer to Figure 24 for a breakdown of the shift count.

Double Shift Only								
6	5	4	3	2	1	0	Bit Position	
64	32	16	8	4	2	1	Shift Value	

Figure 24. Shift Count Breakdown

Each bit position of the shift count represents a shift value, and the sum of the shift value for each bit set in the shift count equals the total number of places shifted.

**NOTE:** The shift value is shown as a decimal value; all references to shift counts in the documentation refer to a decimal count.

If the *jk* field of a left single shift equals  $27_8$  and bits 4, 2, 1, and 0 are set, the shift values would be 16, 4, 2, and 1, respectively. The sum of the shift values would be 23 ( $16 + 4 + 2 + 1$ ); therefore, the instruction would shift left  $23_{10}$  places.

The actual hardware that performs the shifts is the same for both left and right shifts. However, the hardware performs only left shifts. Right shifts are performed according to how data is entered into the shifter, hence the use of two's complement for right shifts.



### Scalar Left Single Shift

Figure 25 is an illustration of how a left single shift is performed for a 054220 instruction ( $S_i S_i < exp$ ). In this example, we shift  $S_2$  left  $jk$  places ( $20_8$ ) with data bit 10 set.

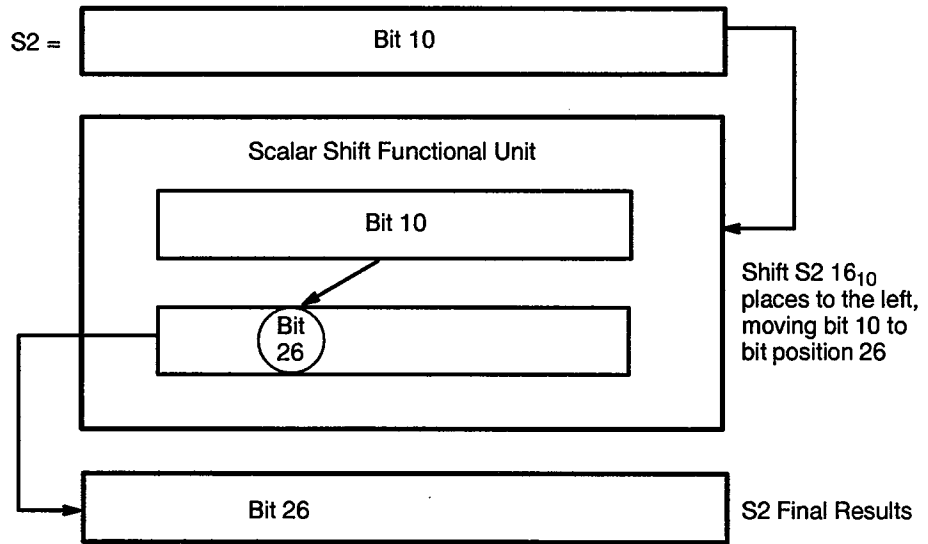


Figure 25. Scalar Left Single Shift

## Scalar Right Single Shift

Figure 26 is an illustration of how a right single shift is performed using left shifts and a two's complement shift count. This example uses a 055254 instruction ( $S_i > S_i \text{ exp}$ ) that shifts  $S_i$  right  $\text{exp} = 100 - jk$  places to  $S_i$ .

In this example, we shift data bit 45 to the right  $24_8$  ( $20_{10}$ ) places. Notice that the  $jk$  field of the instruction 055254 contains  $54_8$ , which is the two's complement of  $24_8$ , causing  $S_2$  to be shifted to the left  $54_8$  places to set bit 25 of the result.

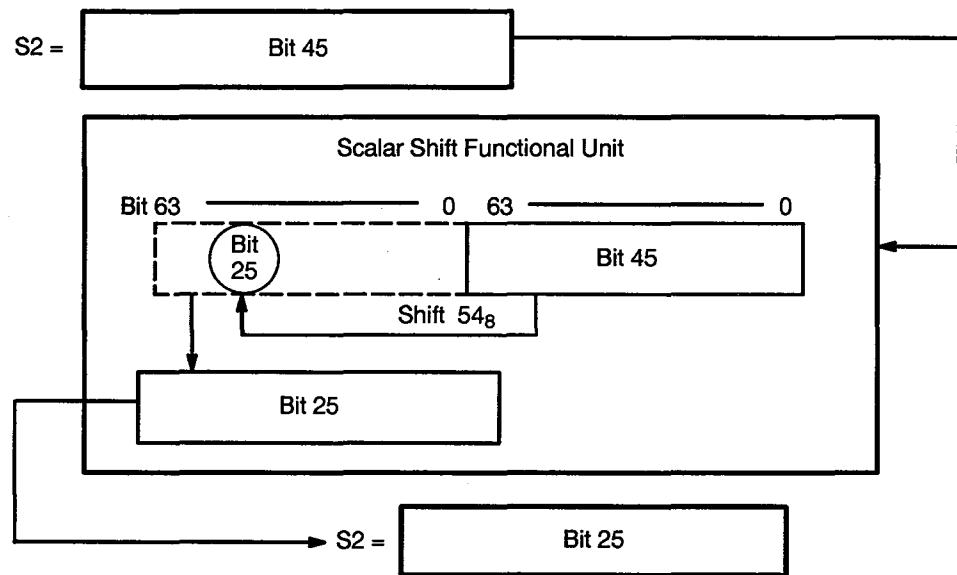


Figure 26. Scalar Right Single Shift

**NOTE:** It is the programmer's responsibility to perform the two's complement of the shift count and supply that value to the functional unit.

## Scalar Left Double Shift

Double shifts are the same as single shifts except that they concatenate two 64-bit registers to form a value. Figure 27 is an illustration of a left double shift using a 056123 instruction ( $S_i, S_j < A_k$ ). In this example, we shift  $S_i$  ( $S_1$ ) and ( $S_j$ ) left ( $A_k$ ) places to  $S_i$ , with  $A_3 = 40_8$  ( $32_{10}$ ),  $S_1$  having bit 30 set, and  $S_2$  having bit 10 set. When a left double shift occurs, the contents of  $S_j$  move into  $S_i$ , and the two registers are positioned as shown with  $S_i$  ahead of  $S_j$ .

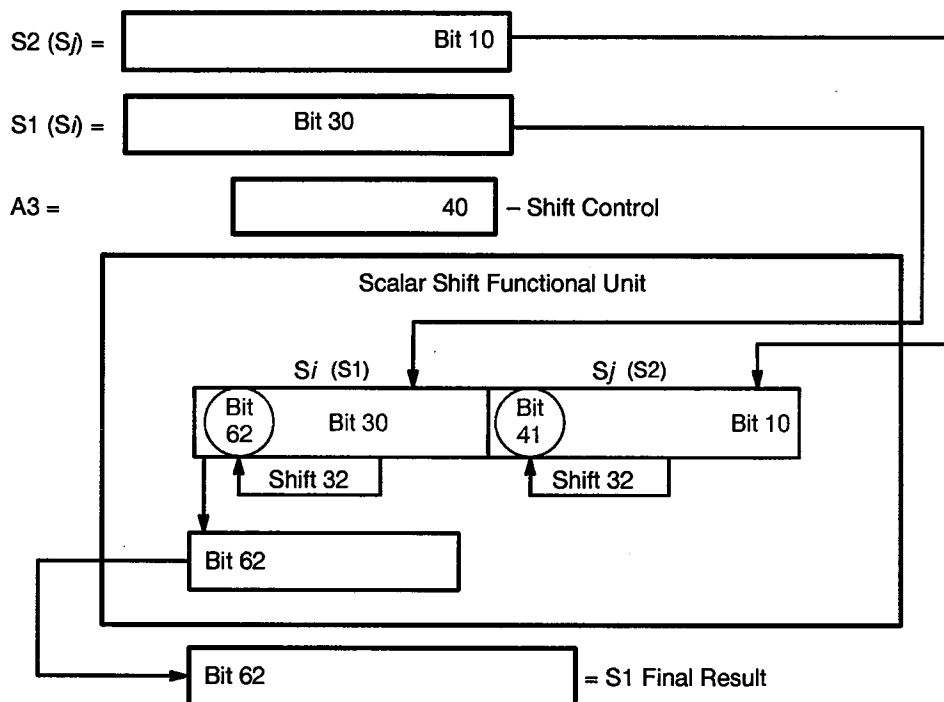


Figure 27. Scalar Left Double Shift

Shifting  $S_i$  and  $S_j$  to the left 32 places puts bit 30 of  $S_1$  at bit position 62 and bit 10 of  $S_2$  at bit position 41. Because bit 41 of  $S_2$  did not make it to the result register  $S_1$ , it is lost. The result bit (bit 62) is then sent to the  $S_i$  ( $S_1$ ) register. The  $S_j$  ( $S_2$ ) register remains unchanged.

## Scalar Right Double Shift

To perform a scalar right double shift, a 057ijk instruction ( $S_i S_j, S_i > A_k$ ) shifts ( $S_j$ ) and ( $S_i$ ) right ( $A_k$ ) places to  $S_i$ . Figure 28 is an illustration of a 057123 instruction with the indicated parameters.

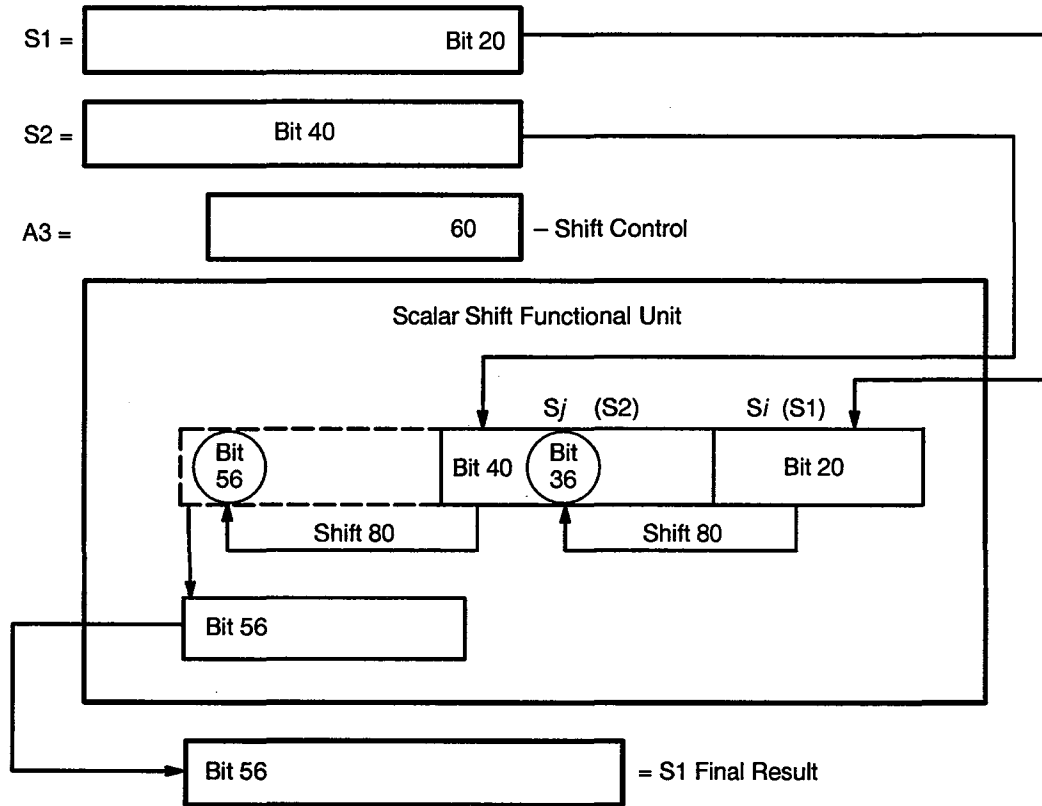


Figure 28. Scalar Right Double Shift

To right shift  $S_j$  and  $S_i$  using left shifts, the two's complement is first performed on  $A_3$ , which currently equals  $60_8$  ( $48_{10}$ ). Because the two's complement is  $120_8$  (or  $1010000_2$  or  $80_{10}$ ), the required shift can be accomplished through successive shifts of  $64_{10}$  and  $16_{10}$  for a total shift of  $80_{10}$  places. A left shift of  $80_{10}$  would move bit 40 of  $S_2$  to bit position 56 inside the dotted box and bit 20 of  $S_1$  to bit position 36 of  $S_2$ . Because bit 36 did not make it into the result register (indicated by the dotted box), it is lost, and bit 56 is sent to the final result.

## Left Single-shift Instruction

Refer to Figure 29 when reading the two following examples of the scalar left single-shift instruction.

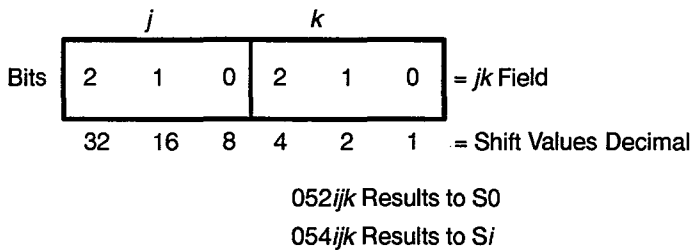


Figure 29. Example of Scalar Left Single-shift Instruction

**Example 1:** Write the instruction to shift S2 left 20<sub>10</sub> places, placing the results into S0.

- Steps:
1. 052*ijk* – left shift instruction result goes to S0
  2. *jk* field– shift count 20<sub>10</sub> = 24<sub>8</sub> = *jk* field
  3. 052224 – final instruction

**Example 2:** Write the instruction to shift S4 left 35<sub>10</sub> places, placing the results into S4.

- Steps:
1. 054*ijk* – left shift instruction result goes to Si
  2. *jk* field– shift count 35<sub>10</sub> = 43<sub>8</sub>
  3. 054443 – final instruction

## Right Single-shift Instruction

---

The right single-shift count is the *jk* field of the instruction, which must either be in the two's complement form or  $100_8$  minus the number of places to right shift. Two examples of a scalar right single-shift instruction follow.

- $053ijk$  results to  $S0$
- $055ijk$  results to  $S_i$

Example 1: Write the instruction to shift  $S5$  right  $10_{10}$  places, placing the results into  $S0$ .

- Steps: 1.  $053ijk$  – right shift instruction results to  $S0$
2. *jk* field – shift count in two's complement equals  $66_8$

$$10_{10} = 12_8 = 001010$$

$$\text{two's complement} = 110101$$

$$\begin{array}{r} + 1 \\ \hline 110110 \end{array} = 66_8$$

3.  $053566$  – final instruction

Example 2: Write the instruction to shift  $S7$  right  $28_{10}$  places.

- Steps: 1.  $055ijk$  right shift instruction results to  $S_i$
2. *jk* field – shift count in two's complement equals

$$28_{10} = 34_8 = 011100$$

$$\text{two's complement} = 100011$$

$$\begin{array}{r} + 1 \\ \hline 100100 \end{array} = 44_8$$

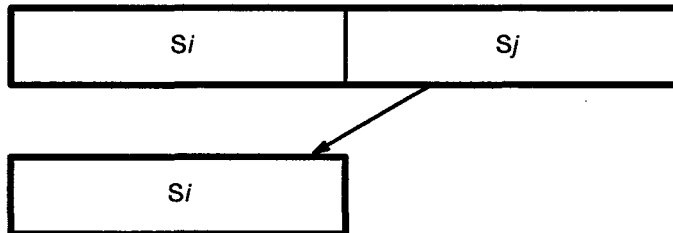
$$\text{or } 100_8 - 34_8 = 44_8$$

3.  $055744$  – final instruction

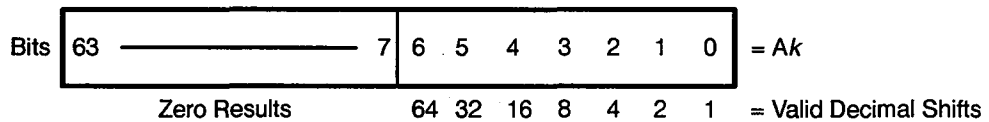
## Left Double-shift Instruction

Refer to Figure 30 when reading the following example of a scalar left double-shift instruction.

056ijk Shift  $S_i$  and  $S_j$  left by  $A_k$  places to  $S_i$



$A_k$  contains the shift count, and A register bits 0 through 6 contain the valid shift counts. If any of bits 7 through 63 are set, the results of  $S_i$  are zeroed.



On a left double shift, the contents of  $S_j$  are always shifted into  $S_i$ . This shift is done inside the scalar shift functional unit.

Figure 30. Example of a Scalar Register Left Double-shift Instruction

Example 1: Write the instruction to left double shift  $S_2$  and  $S_3$   $64_{10}$  places, placing the results into  $S_2$ .

056234 – final instruction, where  $A_4 = 100_8$

**NOTE:** A circular left shift can be effected by issuing a 056 instruction with  $i = j$  and  $(A_k) < 64$ .

## Right Double-shift Instruction

Refer to Figure 31 when reading the following example of a scalar right double-shift instruction.

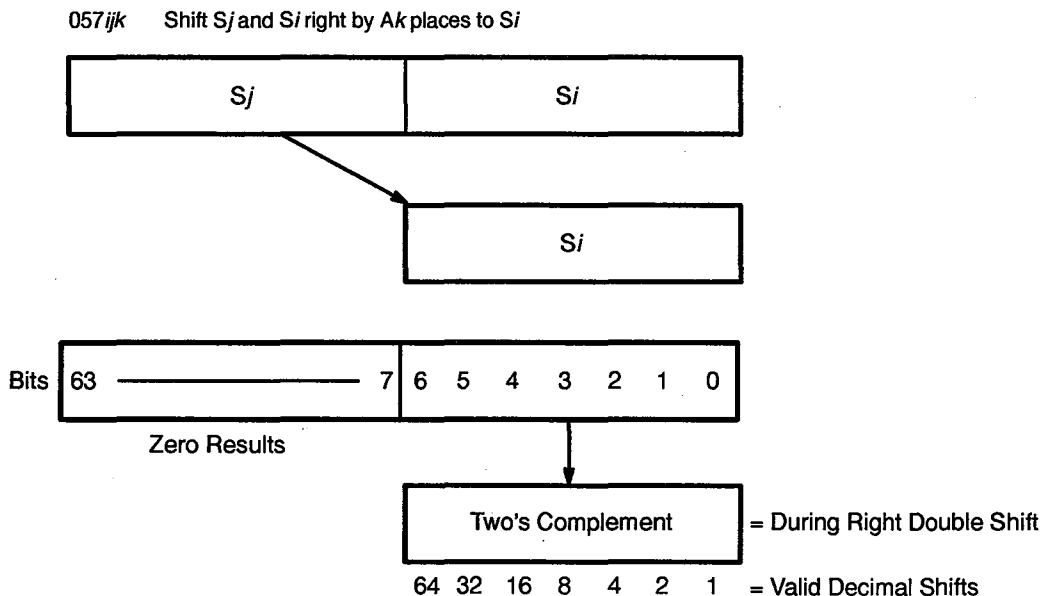


Figure 31. Example of a Scalar Register Right Double-shift Instruction

$A_k$  contains the shift count, and address (A) register bits 0 through 7 contain the valid shift counts. If any of bits 7 through 63 are set, the results of  $S_i$  are zeroed. Also, the hardware generates the two's complement of the shift count on the  $A_k$  register bits 0 through 7 on a right double shift.

On a right double shift, the contents of  $S_j$  are always shifted into  $S_i$ . This operation and the two's complement of the shift count are done inside the scalar shift functional unit.

Example 1: Write the instruction to right double shift  $S_4$  and  $S_5$   $32_{10}$  places, with the results going into  $S_4$ .

057454 – final instruction, where  $A_4 = 40_8$   
hardware generates a shift count of  $140_8$  inside the functional unit.

**NOTE:** A circular right shift can be effected by issuing a 057 instruction with  $i = j$  and  $(A_k) \leq 64$ .



## ADDRESS MULTIPLY

The AN option performs the address multiply operation (a *032ijk* instruction). The AN option also fans out the *Aj* and *Ak* operand used for other A register operations.

When operating in Triton mode, two 48-bit operands are presented to the functional unit to produce a 48-bit result. The AN option then does a sign extension to bit 63 and a leading zero count on the operands to determine whether the results will fit within 48 bits. If the results exceed 48 bits, the 64-bit incompatibility signal sets, causing the Address Multiply Interrupt (AMI) flag to set in the exchange package.

The AN option does not use a standard pyramid formation multiply algorithm. Instead, it uses a variation of the Booth Recode algorithm. This algorithm enables the address multiply unit to reside on a single option.

Half the recode groups are formed immediately upon arrival of the data on the AN option (those groups that are centered on bits 0, 4, 8, 12, 16, etc). One clock period later, using the same logic, those groups centered on bits 2, 6, 10, and 14 are recoded. This method allows a multiply operation to be done on about one-fourth of the logic used in a standard pyramid multiply. Because this method holds the *Ak* operand for 2 clock periods, the AN operand can accept data only every other clock period. Refer to Figure 32 for an illustration of the AN option.

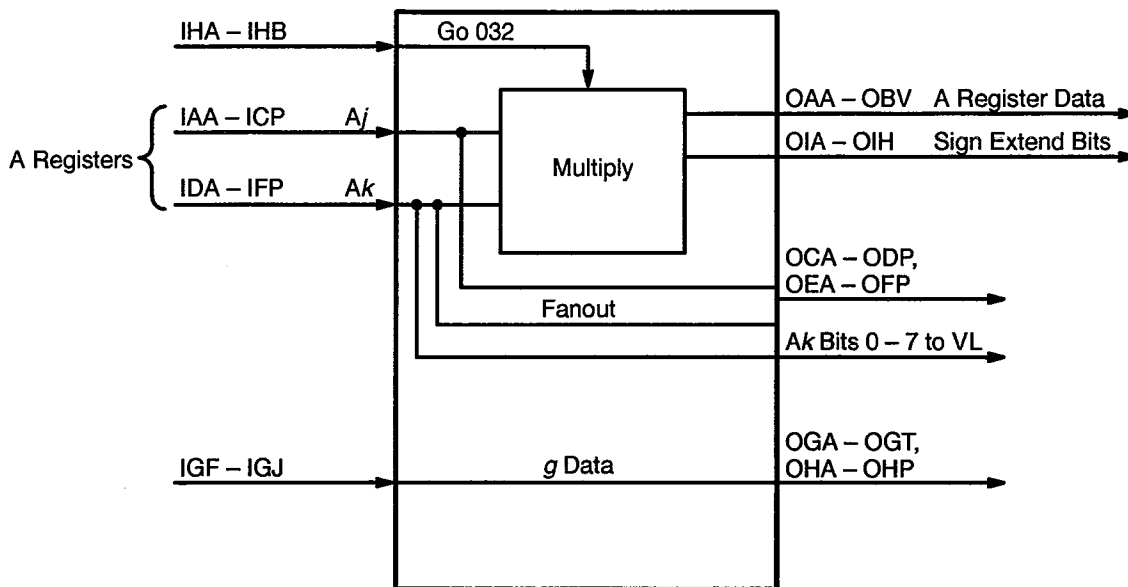


Figure 32. AN Option

### Multiply Algorithm

The multiplier is partitioned into 3-bit recode groups centered on the even bits (0 to 46); a forced zero is added to the first recode group. The recode groups are formed as shown in Table 12, and the following subsections provide examples of standard and Booth Recode multiplication.

Table 12. Recode Groups

Odd Bit	Even Bit	$i-1$	Recode Value	Recode Product
0	0	0	+0	0
0	0	1	+1	$X_{47} - X_0$
0	1	0	+1	$X_{47} - X_0$
0	1	1	+2	$2(X_{47} - X_0)$
1	0	0	-2	$\{2(X_{47} - X_0)\}' + 1$
1	0	1	-1	$(X_{47} - X_0)' + 1$
1	1	0	-1	$(X_{47} - X_0)' + 1$
1	1	1	-0	0
$i-1 = \text{Bit to right of recode group}$		$X_{47} - X_0 = \text{Multiplicand}$		

## Standard Binary Multiplication

Refer to the following example of standard binary multiplication.

$$\begin{array}{r}
 000011 \text{ (3)} \\
 011101 \text{ (35)} \\
 \hline
 000011 \\
 000000 \\
 000011 \\
 000011 \\
 000011 \\
 000011 \\
 000000 \\
 \hline
 0000001010111
 \end{array}$$

## Booth Recode Multiplication

Refer to the following example of Booth Recode multiplication.

$$\begin{array}{r}
 000011 \text{ (3)} \\
 011101 \text{ (35)} \\
 \hline
 000000000011 \\
 11111111010 \\
 00000110 \\
 \hline
 1 \ 00001010111
 \end{array}$$

In the previous example, the multiplier is recoded into bit groups centered on the even bit. A forced zero is appended to the first recode group.

As shown in Table 12, the first recode of the multiplier, bits 1 and 0 and the forced zero, yields a recode value of 010, or +1. In this case, the multiplicand is brought down.

The second recode, bits 3, and 2, and 1 yields a recode value of -1. In this case, a two's complement and a shift of 1 are done on the multiplicand.

The final recode, bits 5, 4, and 3 yields a recode value of +2. This causes a shift of 1 on the multiplicand.

C

C

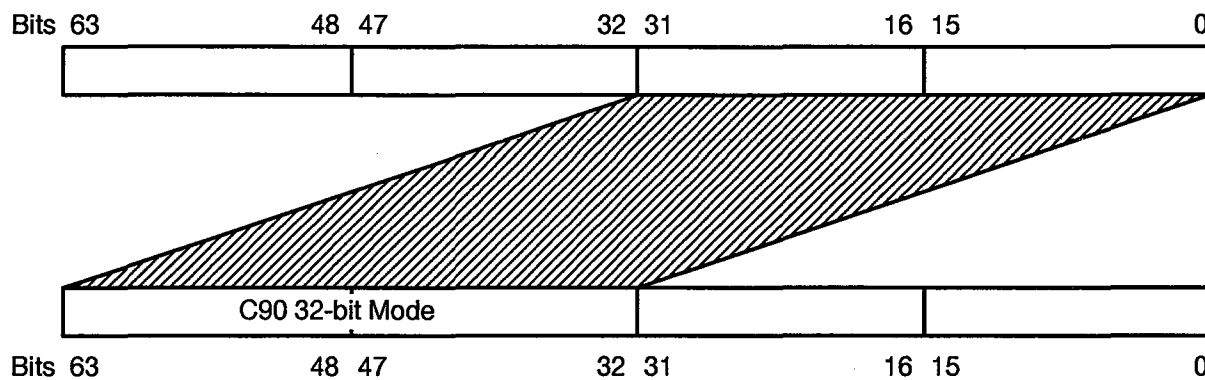
C

## INTEGER MULTIPLY

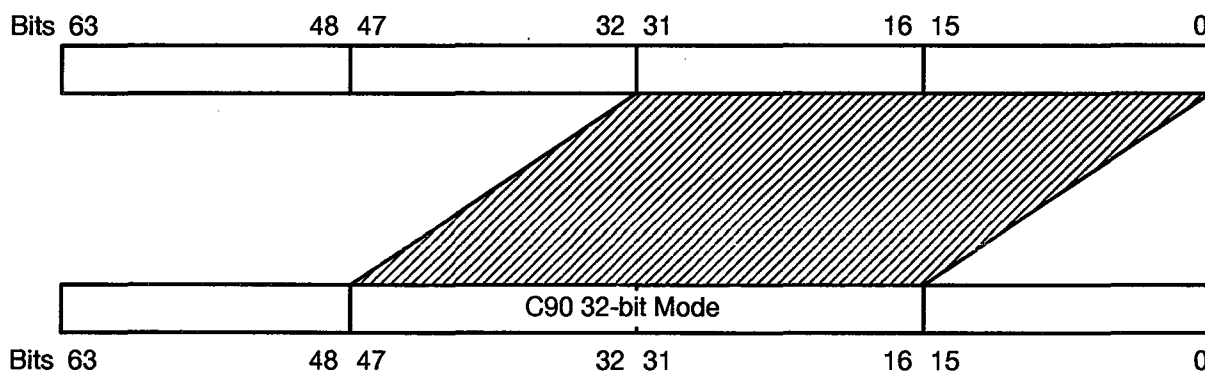
The AM option performs the scalar vector integer multiply operation (*166ijk*). It receives *Sj* and *Vk* operands and produces a 40-bit output to *Vi* for VL length when the system is in Triton mode.

In C90 mode, a 32-bit result forms, and the input operands are modified to produce the 32-bit result. The *Sj* operand must be left shifted 31<sub>10</sub> places, and the *Vk* operand must be left shifted by 16<sub>10</sub> places before executing the *166ijk* instruction, as shown in Figure 33.

The AM option, like the AN option, also uses the Booth Recode algorithm for the multiply operation. The AN option also does a leading zero count on the operands to determine whether the results will fit within 40 bit positions. The input operands are passed through the floating-point multiply unit before they arrive at the AM option, as shown in Figure 34.



Sj bits 0 through 31 are gated into bit positions 32 through 63 for C90 mode.



Vk bits 0 through 31 are gated into bit positions 15 through 47 for C90 mode.

Figure 33. C90 Operation Mode

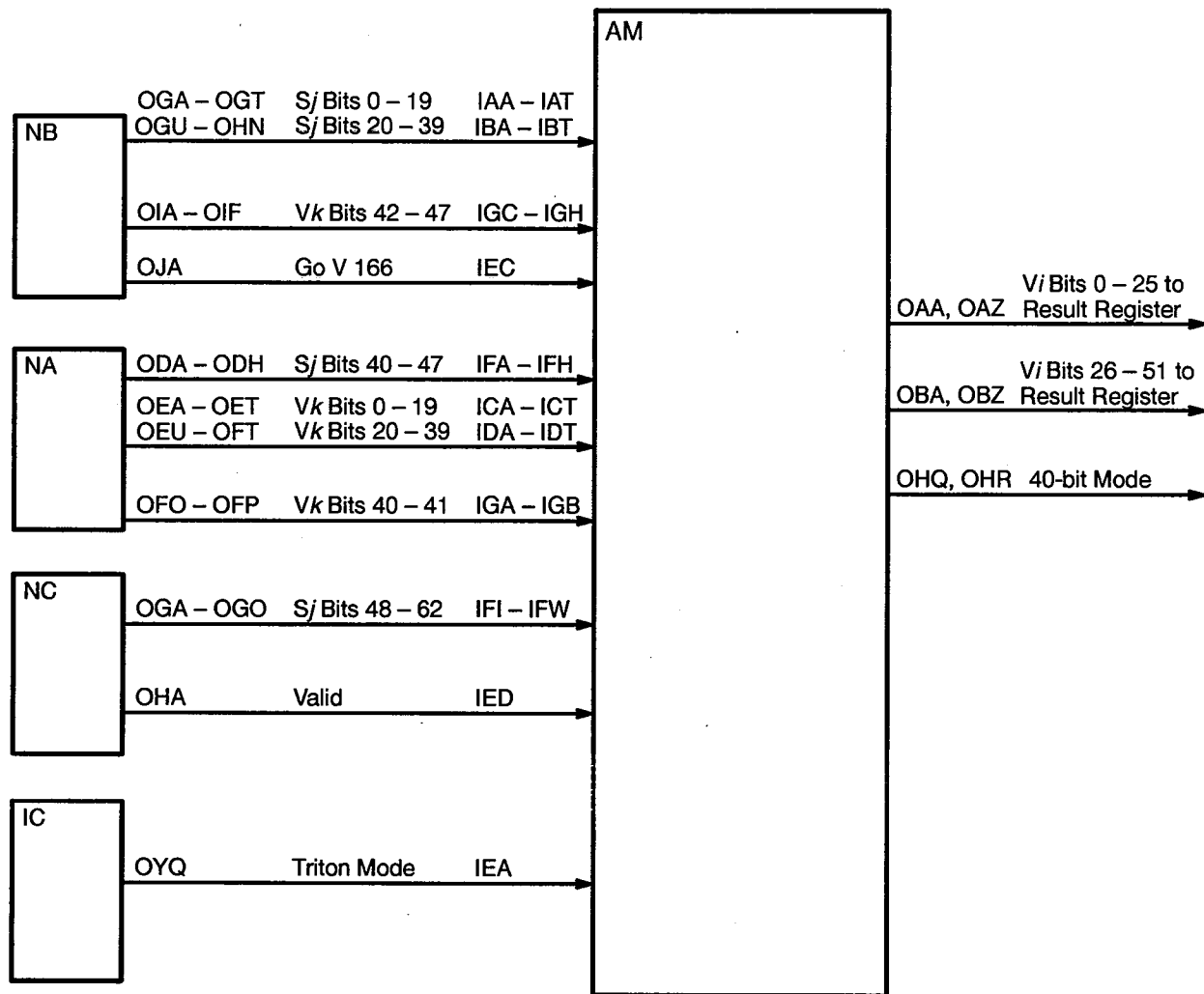
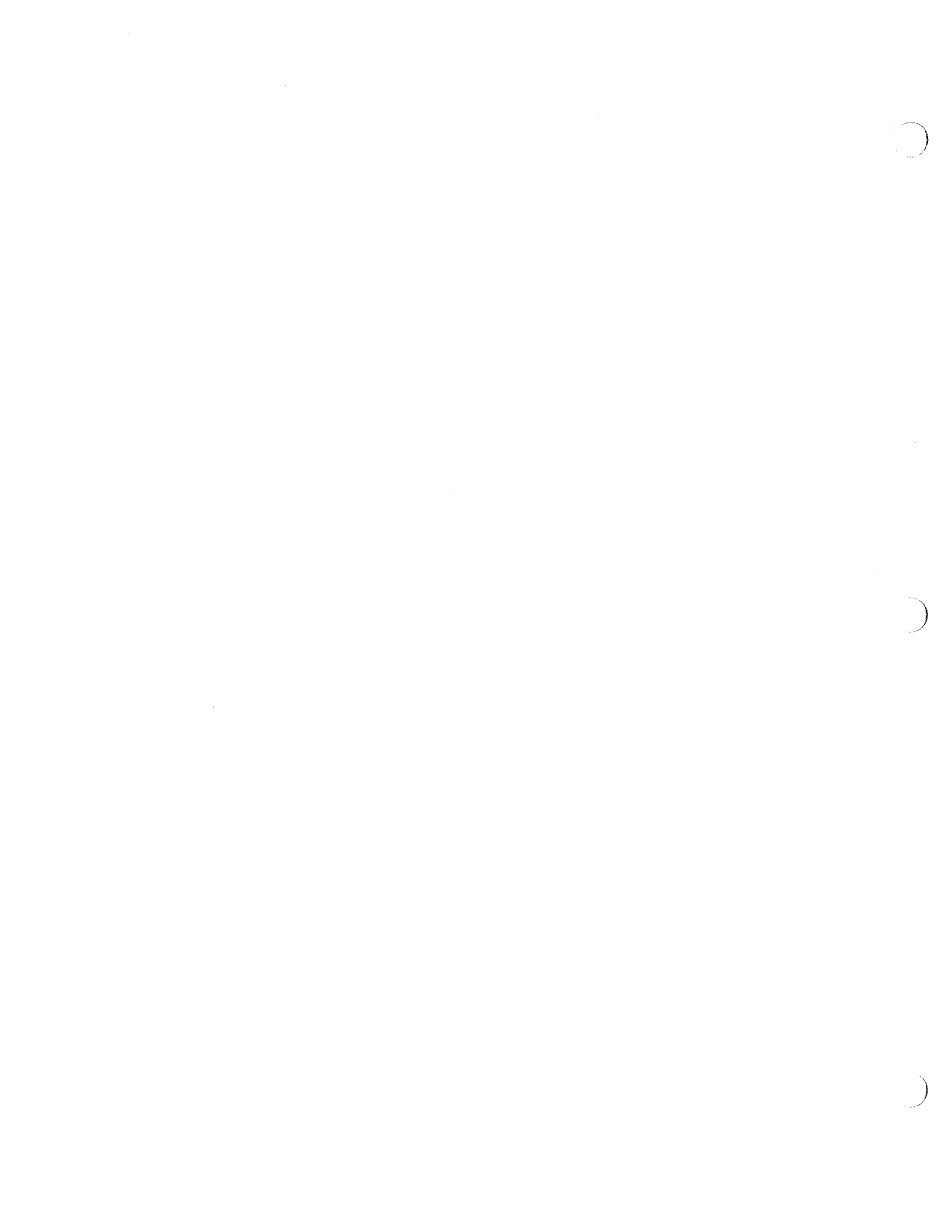


Figure 34. AM Option Inputs





# VECTOR REGISTERS

A CRAY T90 series computer system contains eight vector (V) registers, which are designated V0 through V7. Each register contains  $128_{10}$  elements; each element is  $64_{10}$  bits wide. The  $128_{10}$  elements are divided into two pipes of even and odd elements.

The vector registers have their own integer functional units, which include vector add, vector logical 1, vector logical 2, vector shift, vector population, vector leading zero count, and 32-bit integer multiply. The vector registers share the floating-point functional units with the scalar registers. The floating-point functional units include floating-point add, floating-point multiply, floating-point reciprocal and bit matrix multiply.

The vector registers can send data to memory or load data from memory. The number of elements sent to a functional unit (including memory) depends on the value of the vector length (VL) register. Any element of a vector register can be loaded into a scalar register, and any scalar register can be loaded into any element of a vector register by using the *076ijk* and *077ijk* instructions.

The vector registers use 1-parcel instructions. In a 1-parcel instruction, the *gh* field contains the instruction decode, and the *ijk* field contains the operands and destination. The *gh* field of the instruction indicates the functional unit needed, and the *ijk* field indicates the vector registers used. Generally, the *k* field of the instruction contains the vector operand registers V0 through V7. The *j* field of the instruction can be either *Sj* or *Vj*, depending on the instruction. The *i* field of the instruction is used as the destination or result register.

Some vector instructions, when preceded by a 005400 instruction, cause the instruction to execute in Triton mode as opposed to C90 mode of operation. If, for example, an instruction sequence of 005400 150*ij*0 issues, a left shift of  $V_j V_0$  places to  $V_i$  is performed. If the 005400 instruction had not preceded the 150*ij*0 instruction, a left shift of  $V_j A_0$  places to  $V_i$  would have occurred.

The vector registers in the Triton system contain a dual set of functional unit pipes. Each functional unit has another identical functional unit. For example, the vector add functional unit is duplicated so that all the even elements go to one of the vector add functional units, while all the odd elements go to the other vector add functional unit. The even and odd elements are sent to the functional unit simultaneously, and the two results are loaded back into the result vector register simultaneously.

If the vector add functional unit fails in the even elements, the cause of the failure is the pipe 0 vector add. Pipe 1 handles the odd vector elements. If the vector length register is an even value, the results are written into the vector register simultaneously using pipe 0 and pipe 1, until the last element specified by the vector length is used. Refer to Table 13 for a list of the vector register options.

Table 13. Vector Register Options

Option Type	Number Used	Description
VA	2	Provide read/write address and control (VA0 pipe 0) (VA1 pipe 1) Vector length register Functional unit release
VF	4	Pipe control (VF0,VF1 for pipe 0) (VF2,VF3 for pipe 1)
VM	16	Data multiplexing (VM0 – VM7 pipe 0) (VM8 – VM15 pipe 1) Vector add functional unit Vector logical functional unit
VR	16	Data multiplexing and storage (VR0 – VR7 pipe 0) (VR8 – VR15 pipe 1)

## VA Option

---

The VA option provides vector read and write control. There are two VA options on a CPU: VA0 provides address and control for the even elements of the vectors, and VA1 provides the address and control for the odd elements. The VA options have the following common functions:

- Vector read and write address
- Read and write vector length
- Vector chaining control

The VA options also have the following unique features:

- VA0
  - Release vectors for write operations
  - Functional unit release for:
    - Vector logical #1
    - Vector shift
    - Vector floating-point multiply
    - Vector reciprocal
  - Even-element addressing
- VA1
  - Release vectors for read operations
  - Functional unit release for:
    - Vector logical #2
    - Vector adder
    - Vector floating-point add
    - Vector matrix multiply
  - Odd-element addressing

## Vector Length Register

The vector length register is located on the VA option. There are two VA options, one for each pipe. Both vector length registers are loaded with *Ak* data bits 00 through 06 from the AR000 option. These bits are needed to achieve values from 0 to 177<sub>8</sub>. If a value of all 0's is entered, the VL register is forced to a value of 200<sub>8</sub>.

When the vector length value is entered, it is entered into a countdown register. VL bit 0 is removed so a VL value of 200 will be a value of 100 in the active register (a pseudo right shift). This is done because each pipe handles only 100 elements. Every time VL decrements, it generates the **Advance Address** signal. The VA option also checks VL bit 0 to determine whether the vector length is odd or even. This enables either pipe 0 for odd vector lengths, or pipe 1 for even vector lengths, on the last operation.

## Chaining

If  $V_i$ ,  $j$ , or  $k$  is reserved as a destination and the next instruction wants to use the same vector register as an operand, the next instruction is allowed to issue. This is referred to as chaining.

Chain slot time is the time required for the result of a previous instruction to be presented to the inputs on the VR options. If another instruction is waiting for these results or is addressing the same element, the VR option passes the results directly to the read-out register. The VA option controls the vector chaining by controlling the issuing of the **Go Write** signal.

Chaining to common memory read operations occurs on 8-word boundaries. Vector control waits for 8 contiguous words to become valid before the read of that group is allowed.

## VF Option

---

There are four VF options on the CP module. VF0 and VF1 control fanout for pipe 0; VF2 and VF3 control fanout for pipe 1. The VF options perform the following functions.

- Instruction parcel data fanout to VR options
- Vector add carry and enable summations and bit toggles
- Vector register parity error information
- Vector functional unit delay chains
- Vector functional unit data valids
- $V_k$  address buffering for common memory
- Release of  $V_i$  for write operations

## VM Option

---

The VM options perform write data multiplexing on an 8-bit slice of all functional unit data. There are 16 VM options. VM000 to VM007 are for even-element steering, and VM008 to VM015 are for odd-element steering.

The VM option performs the following functions:

- Read and write data steering
- Vector read-out control
- Vector add functional unit
- Both vector logical functional units

## VR Option

---

A total of 16 VM and VR options reside on the CP module as shown in Table 14. Each option performs read data steering and also vector data storage. The contents of the selected vector register are gated to one of the following destinations; the read data steering is done on 4-bit slices.

- Floating-point add
- Floating-point multiply
- Reciprocal, pop, parity, LZ
- Shift
- Common memory port A
- Common memory port B
- Common memory port C
- Common memory write data
- V data to scalar
- Bit matrix multiply

The VM and VR options contain four high-speed register (HSR) storage arrays that are 18 bits wide by 64 elements deep. Sixteen of the bits are data and 2 bits are for parity. VR000 through VR007 store vector data for the even elements (pipe 0), and VR008 through VR015 store data for the odd elements (pipe 1).

**NOTE:** VM/VR options 12 through 15 do not handle exchange data.

Table 14. VM/VR Data Steering

Option Pipe 0/Pipe 1	VM3/11	VR3/11	VM2/10	VR2/10	VM1/9	VR1/9	VM0/8	VR0/8
Read Bits	28 – 31	24 – 27	20 – 23	16 – 19	12 – 15	8 – 11	4 – 7	0 – 3
Write Bits	24 – 31	–	16 – 23	–	8 – 15	–	0 – 7	–
Exchange Bits	60 – 63	55 – 59	52 – 55	48 – 51	44 – 47	40 – 43	36 – 39	32 – 35
Option Pipe 0/Pipe 1	VM7/15	VR7/15	VM6/14	VR6/14	VM5/13	VR5/13	VM4/12	VR4/12
Read Bits	60 – 63	56 – 59	52 – 55	48 – 51	44 – 47	40 – 43	36 – 39	32 – 35
Write Bits	56 – 63	–	48 – 55	–	40 – 47	–	32 – 39	–
Exchange Bits	28 – 31	24 – 27	20 – 23	16 – 19	12 – 15	8 – 11	4 – 7	0 – 3

Each VR option has an input that is used to force parity errors into the HSR arrays. The maintenance channel provides the following two features: force RAM parity error internal (code 100) and force RAM parity error external (code 140). Through the use of the maintenance channel, a specific loop controller and a specific chip can be given a maintenance function such as force parity error.

## Write Data Steering

---

The VM options receive the *i* instruction field from the VF options; this field performs internal gating of data to the correct register. The *i* field and the instruction decode enable separate write paths for each vector. This path stays selected until a new instruction issue changes it. All the write paths are separate and all can be active at the same time. Refer to Figure 35 for an illustration of the write data path.

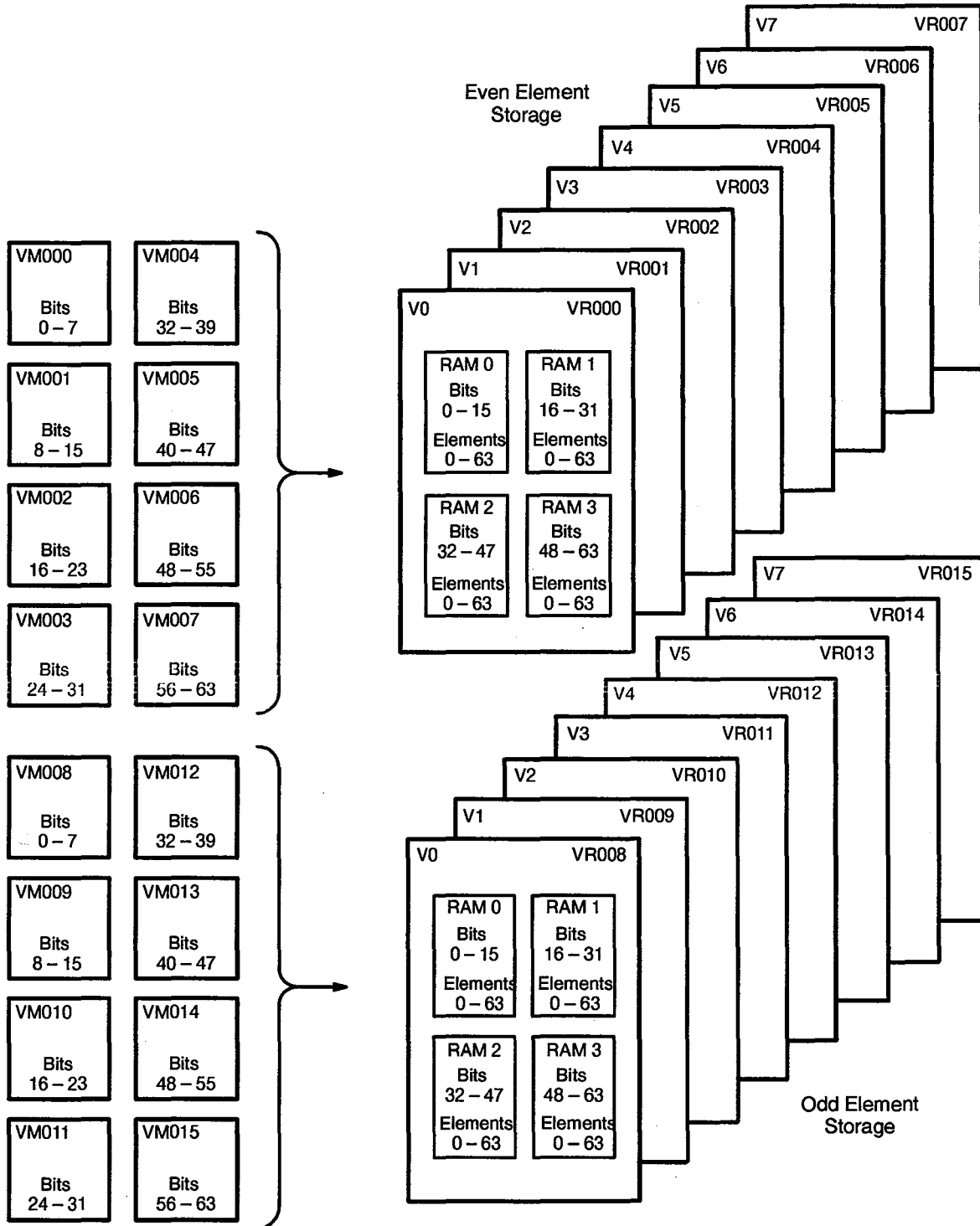


Figure 35. Write Data Path

## Read Data Steering

---

Both the VM and the VR options are responsible for read data steering. Each VM and VR option steers 4 bits for all eight vector registers to one of the following destinations:

- Floating-point add
- Floating-point multiply
- Reciprocal, pop, parity, leading zero
- Shift
- Common memory port A, B, C
- V data to scalar

The VM and VR options receive the  $j$  and  $k$  fields of the instruction from the VF option along with the instruction; this enables one of eight vector paths to which data is steered. These paths stay selected until another instruction changes them. All the read paths are separate and all can be active at the same time. Figure 36 shows the read data path for pipe 0 and pipe 1 (even elements), and Figure 37 shows the read data path for pipe 0 and pipe 1 (odd elements). Refer also to the following diagrams for additional related vector register information:

- Figure 38 – vector register write block diagram (pipe 0)
- Figure 39 – vectors 0 through 3 pipe 0/1 read data path
- Figure 40 – vectors 4 through 7 pipe 0/1 read data path
- Figure 41 – vectors 0 through 3 pipe 0/1 write data path
- Figure 42 – vectors 4 through 7 pipe 0/1 write data path
- Figure 43 – vector register decode bit fanout (pipe 0 and 1 path 1)
- Figure 44 – vector register decode bit fanout (pipe 0 and 1 path 2)
- Figure 45 – S register to vectors
- Figure 46 – memory data to vectors (even elements)
- Figure 47 – memory data to vectors (odd elements)



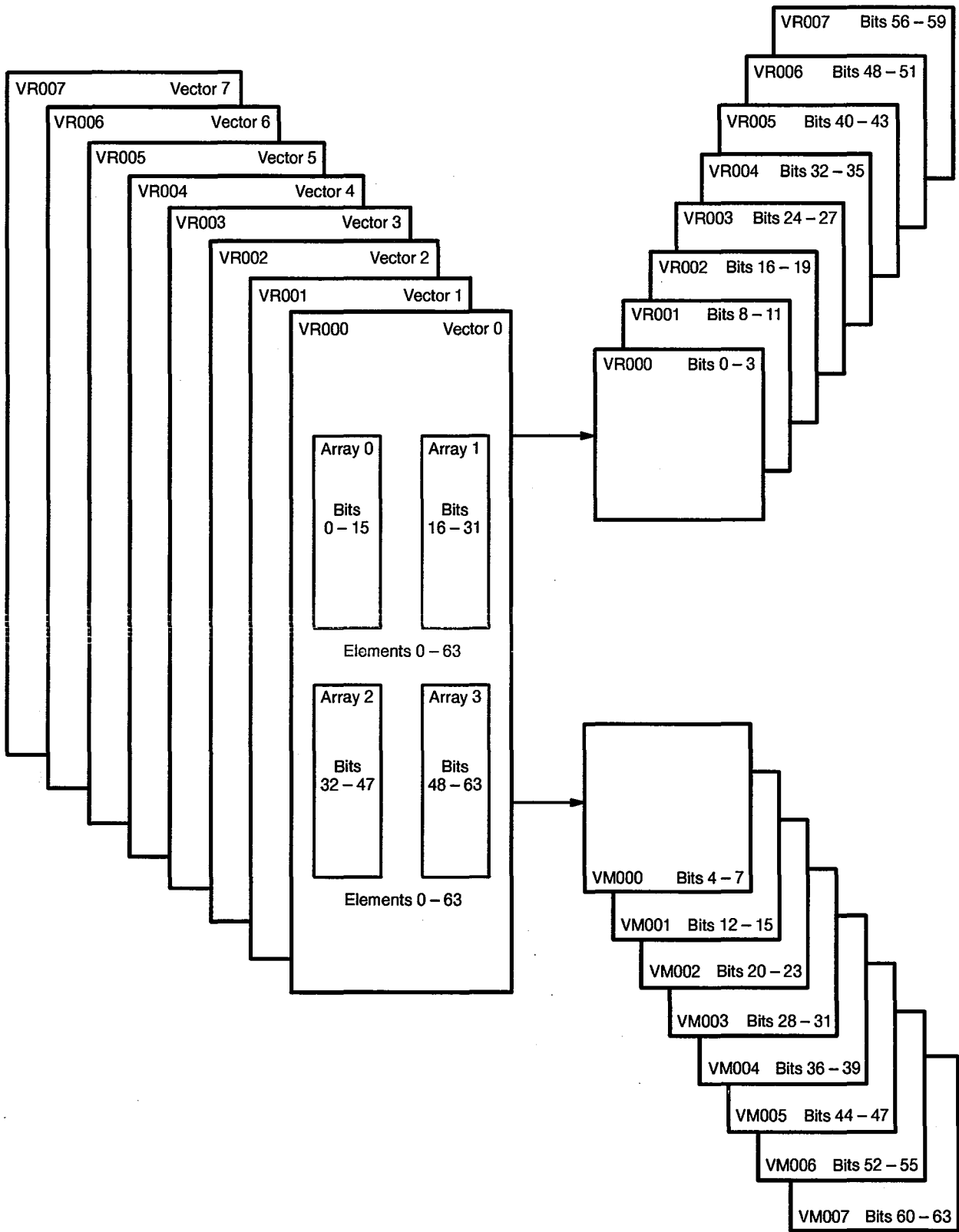


Figure 36. Read Data Path for Pipe 0 (Even Elements)

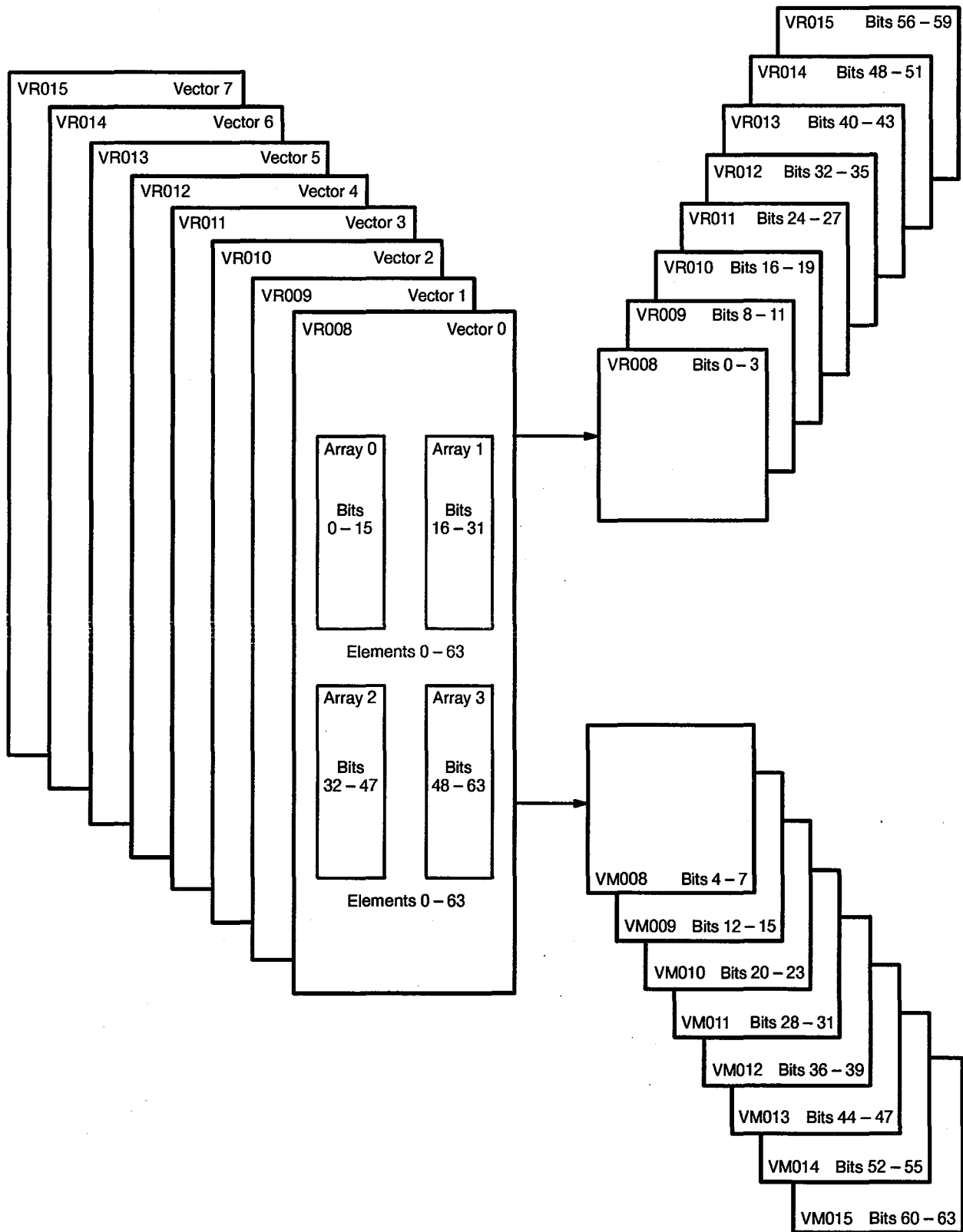


Figure 37. Read Data Path for Pipe 1 (Odd Elements)

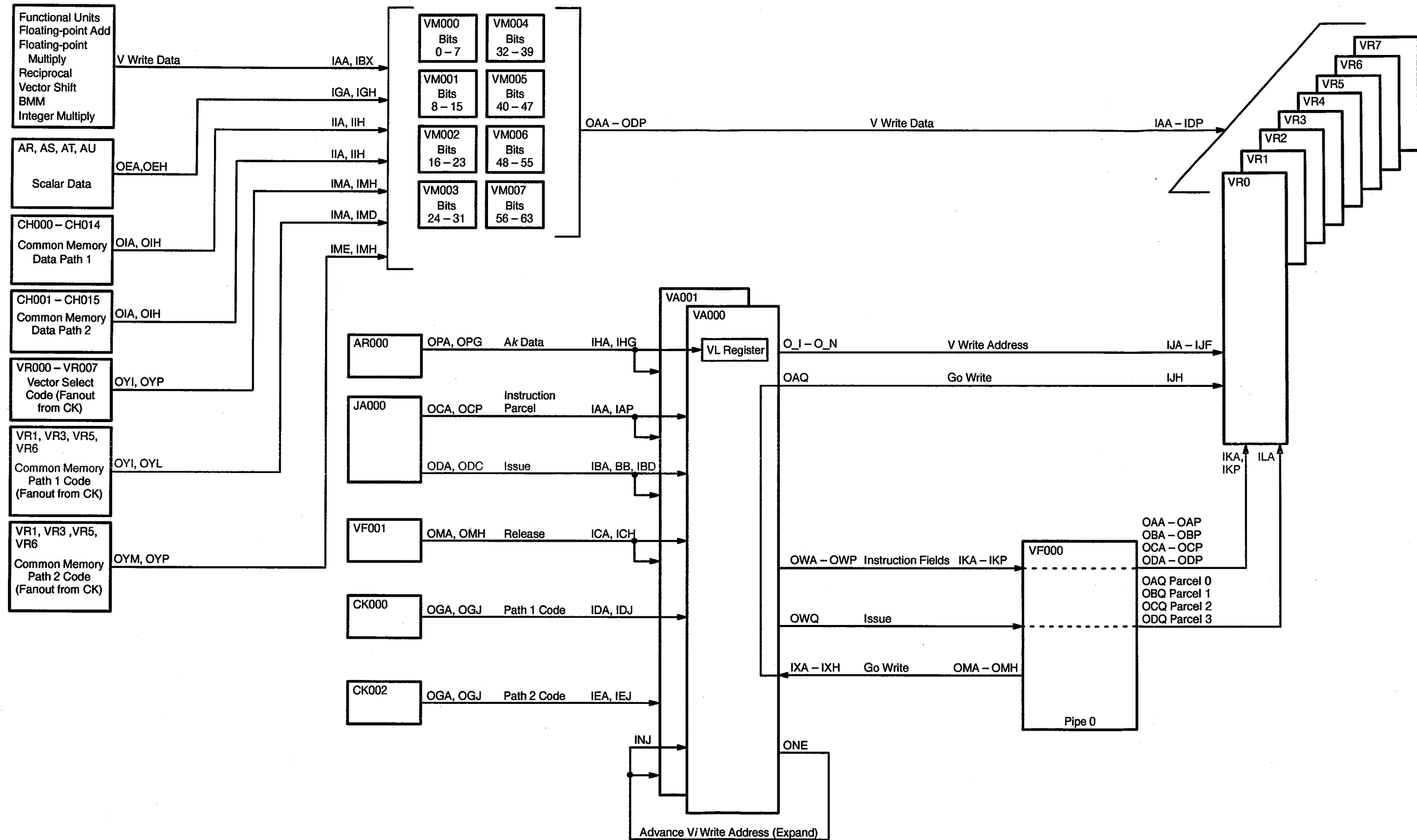


Figure 38. Vector Register Write Block Diagram (Pipe 0)

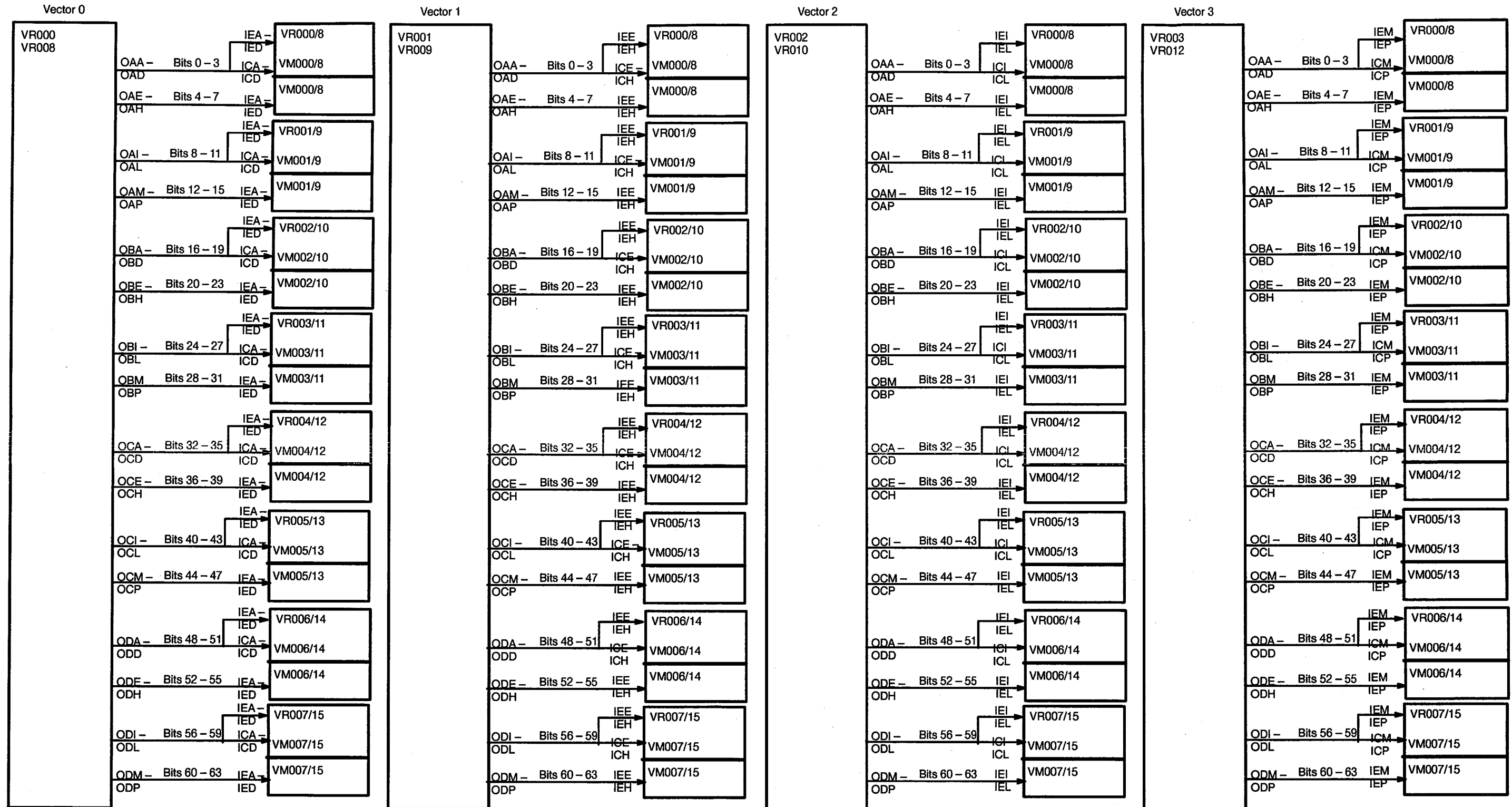


Figure 39. Vectors 0 through 3 Pipe 0/1 Read Data Path

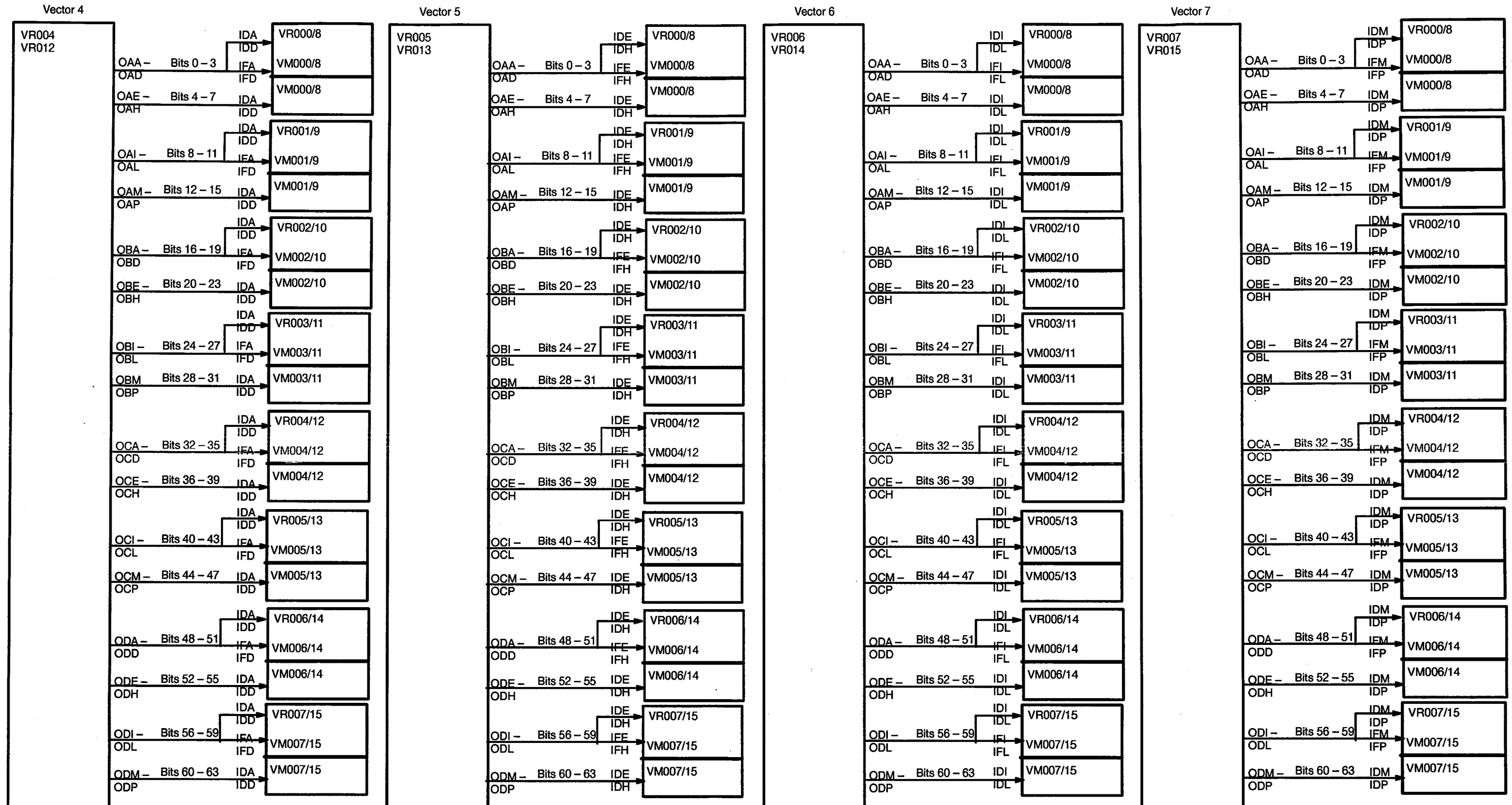


Figure 40. Vectors 4 through 7 Pipe 0/1 Read Data Path

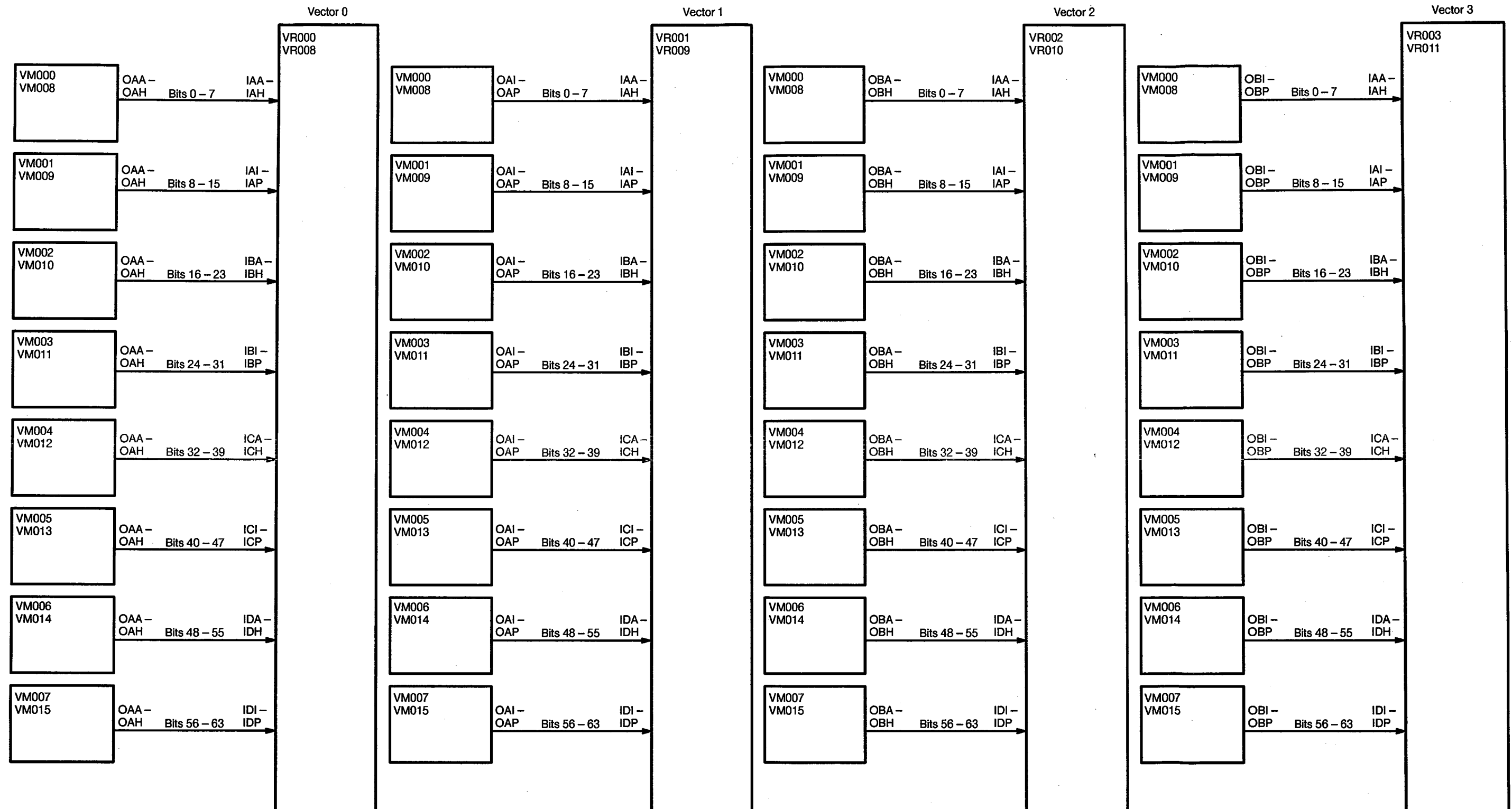


Figure 41. Vectors 0 through 3 Pipe 0/1 Write Data Path

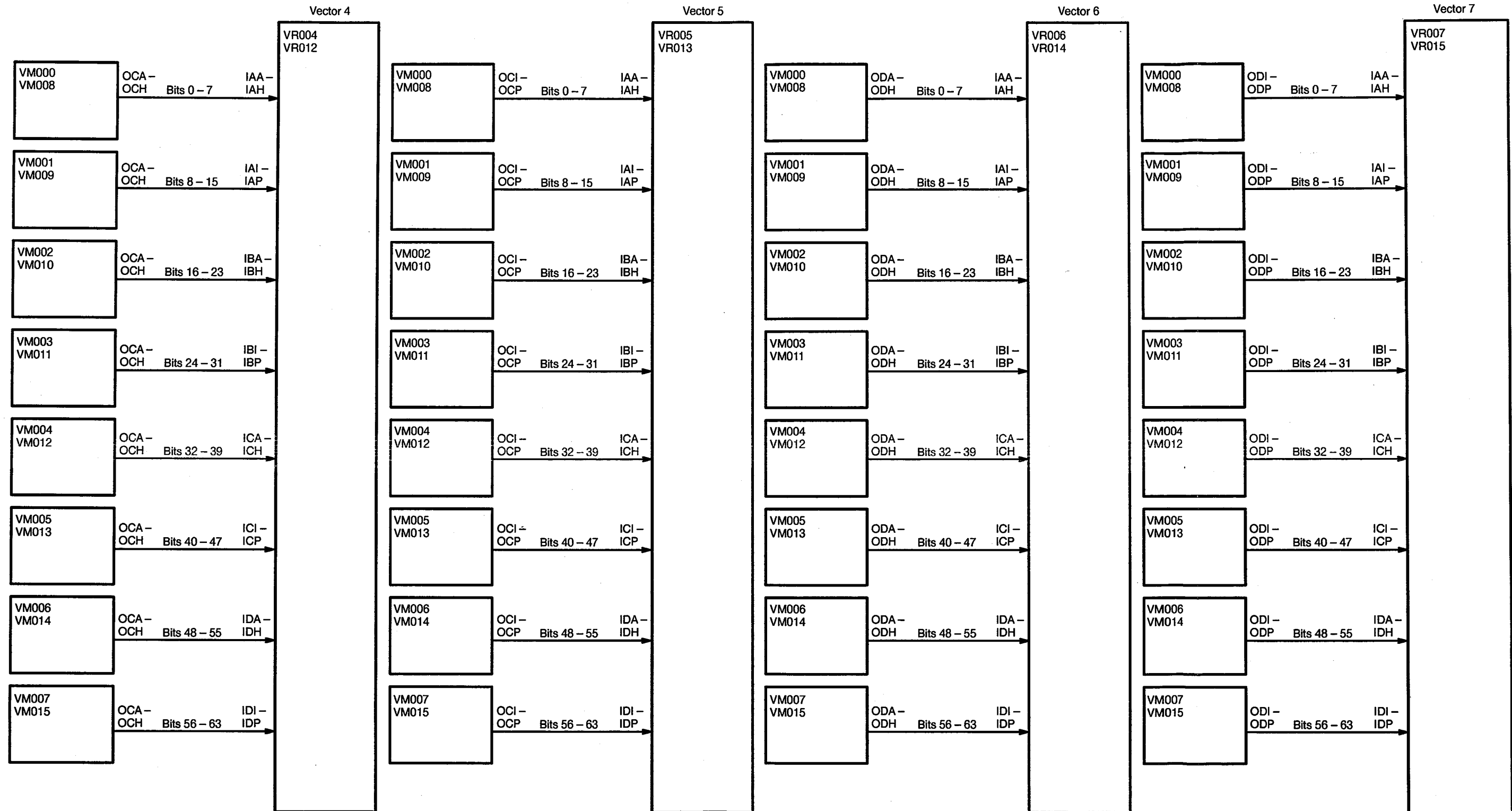


Figure 42. Vectors 4 through 7 Pipe 0/1 Write Data Path

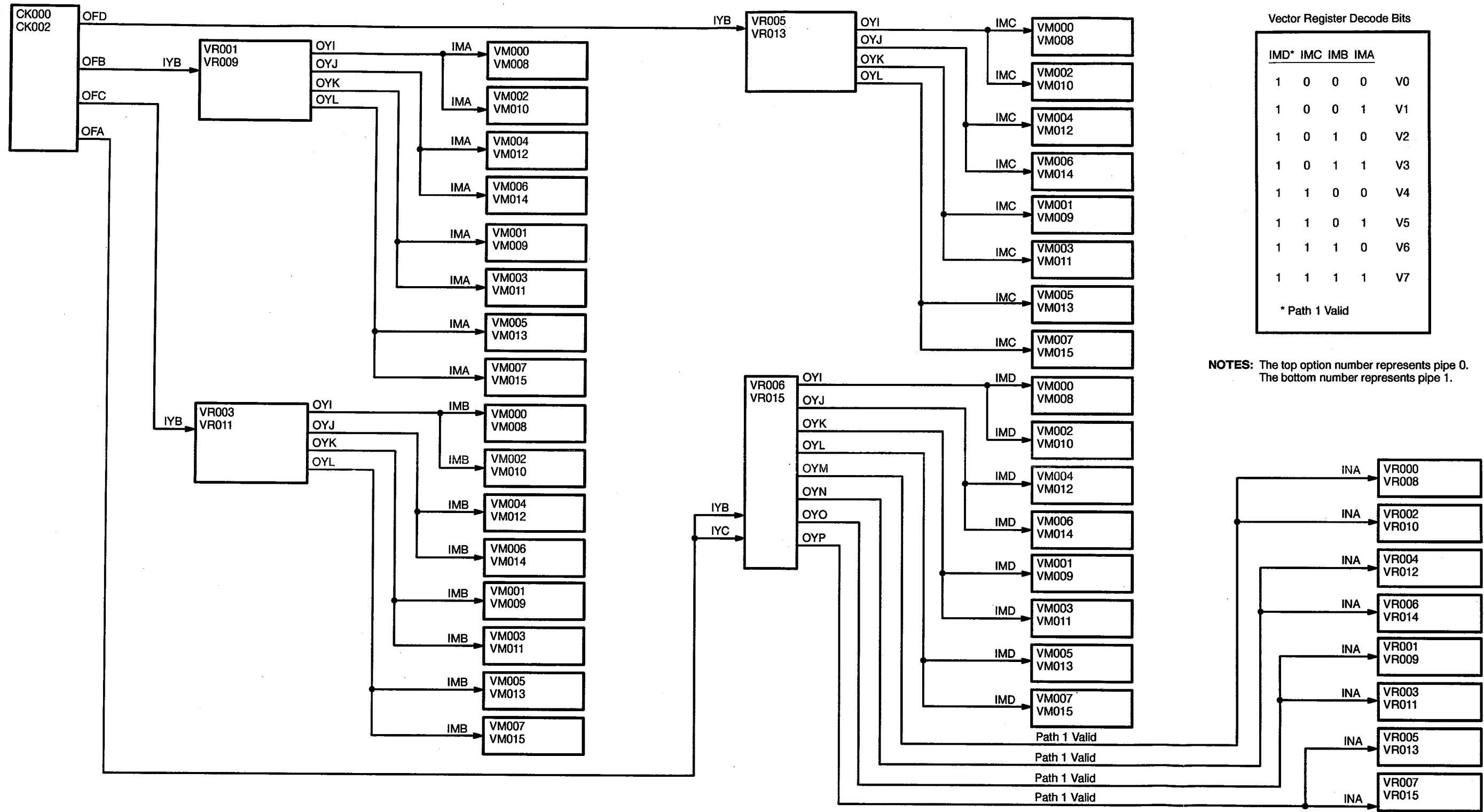


Figure 43. Vector Register Decode Bit Fanout (Pipe 0 and 1 Path 1 Only)



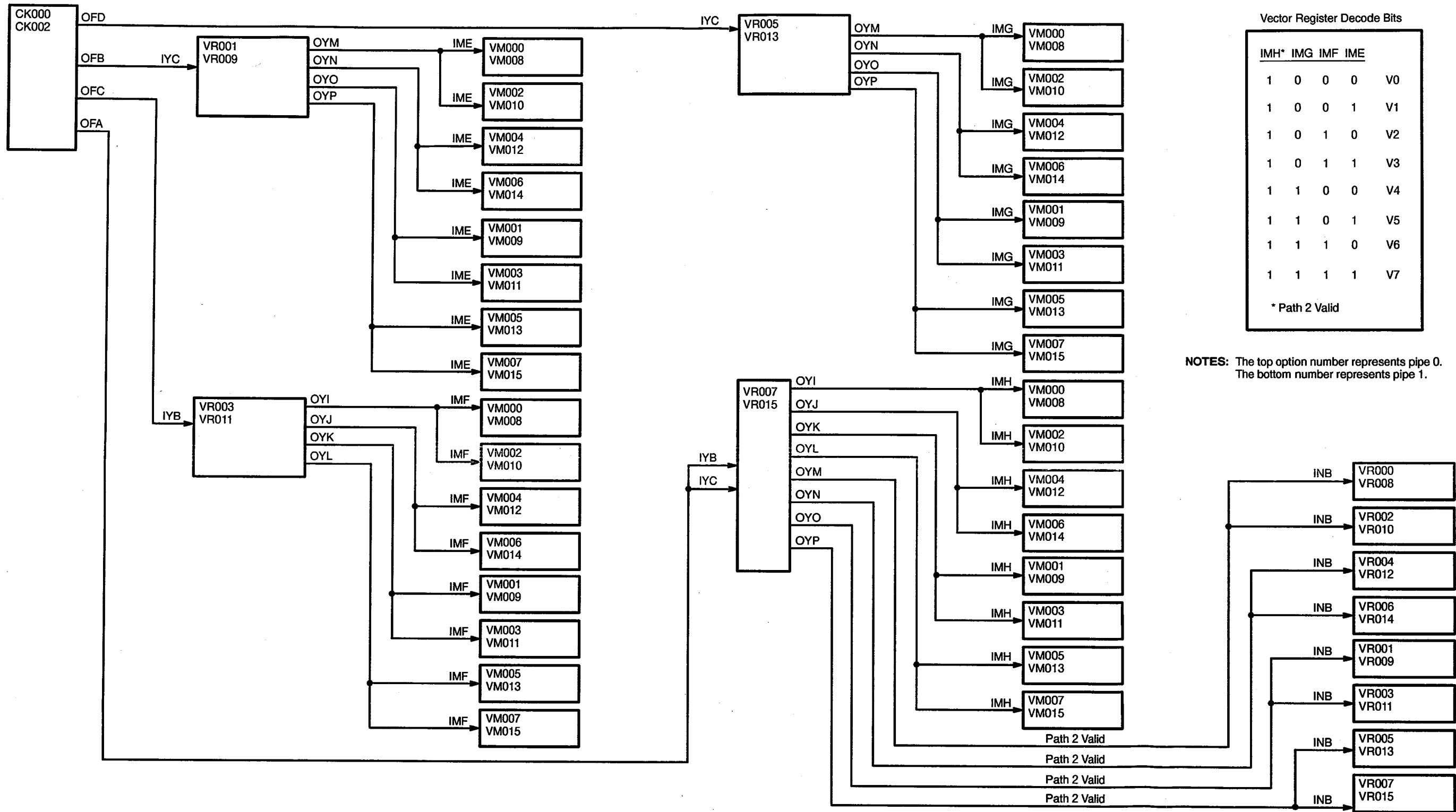
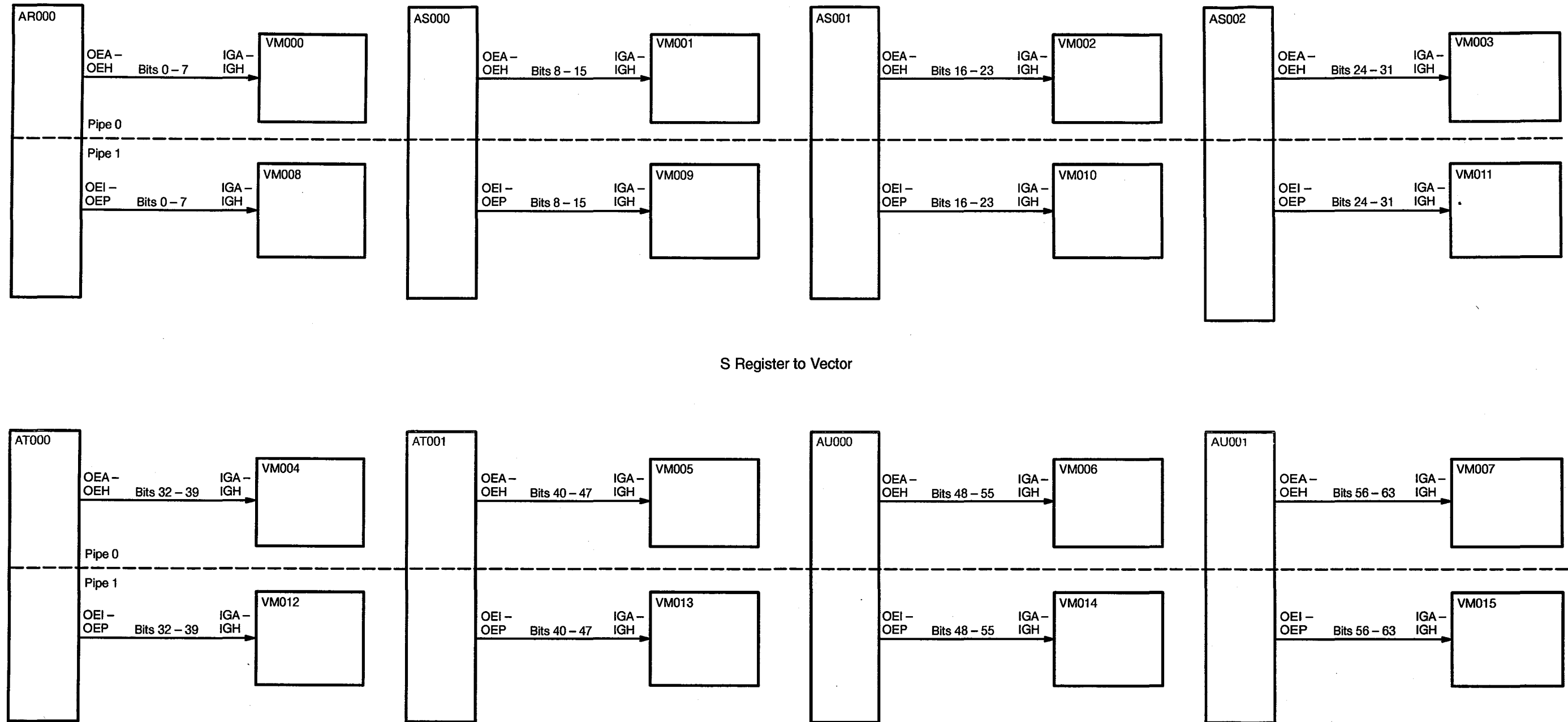
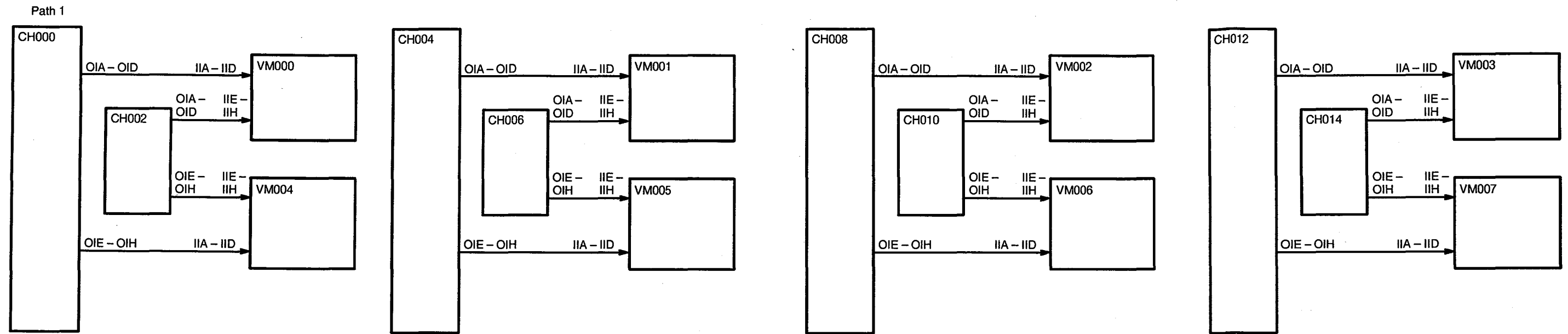


Figure 44. Vector Register Decode Bit Fanout (Pipe 0 and 1 Path 2 Only)



S Register to Vector

Figure 45. S Register to Vectors



Common Memory Data to Vector Paths 1 and 2 Even Elements

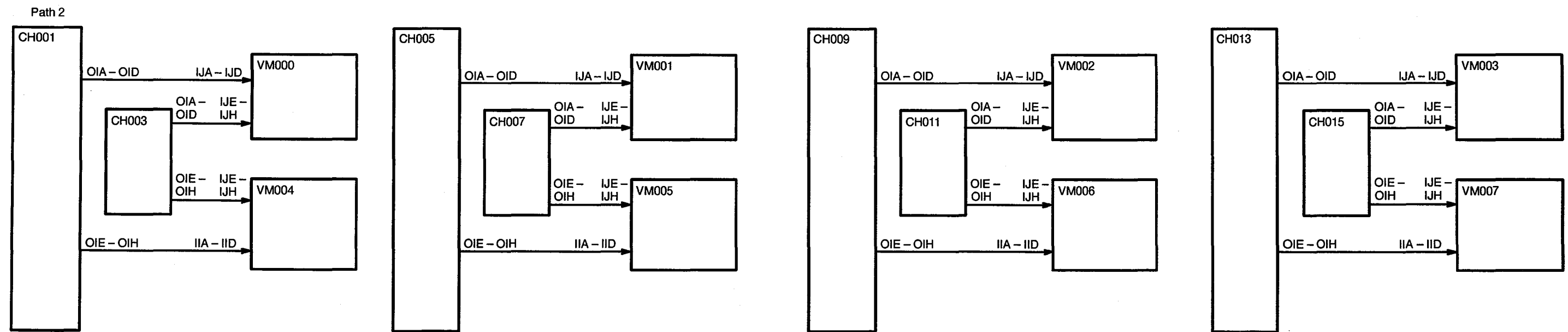
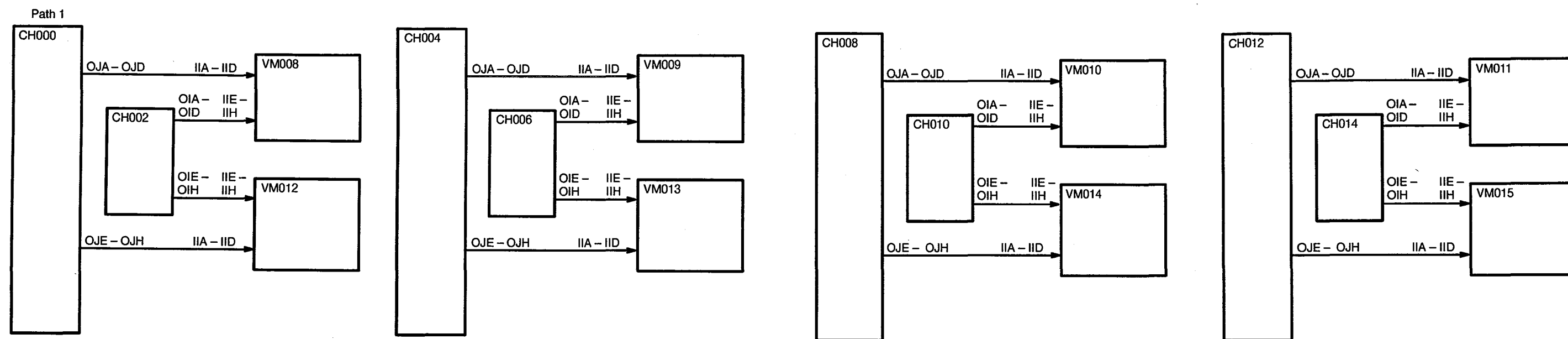


Figure 46. Memory Data to Vectors (Even Elements)



Common Memory Data to Vector Paths 1 and 2 Odd Elements

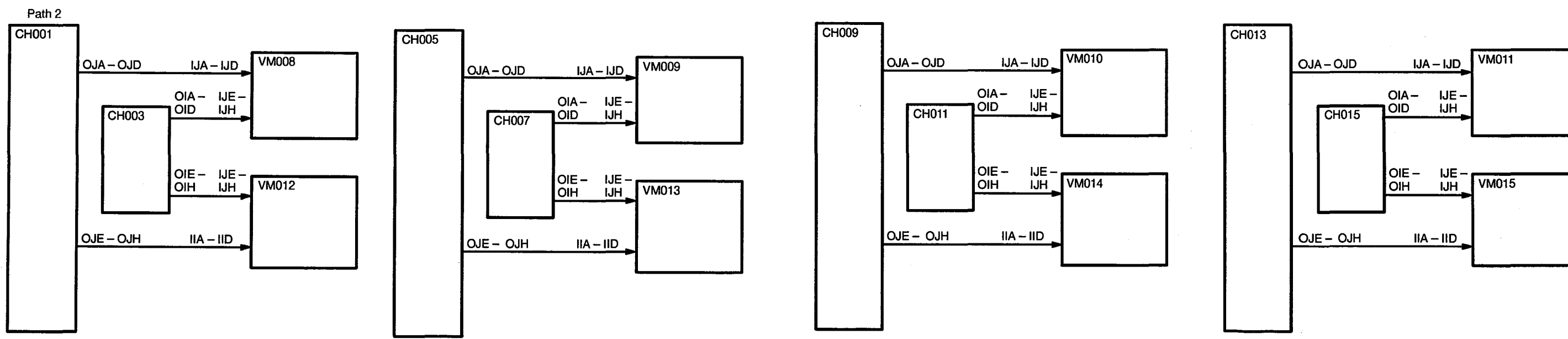


Figure 47. Memory Data to Vectors (Odd Elements)

# VECTOR LOGICAL

Refer to Figure 48 for a vector logical block diagram. There are two vector logical units in a CRAY T90 series system; each unit operates independently. These functional units reside on 16 VM options. VM000 through VM007 handle pipe 0 (the even elements), and VM008 through VM015 handle pipe 1 (the odd elements). Each VM option operates on a 4-bit slice of all eight vector registers.

The vector logical units receive data from the VR options and send the results back to the vector registers. The second vector logical unit is enabled by setting mode bit 2 (ESL) in the mode field of the exchange package. When both logical units are enabled, data is first processed in the second unit. This is done because only the first unit can process the 146 and 147 (vector merge) instructions. For example, if a 140 instruction (logical product) issues, the second unit processes the instruction in case a 146 or 147 issues next. If the first unit processed the 140 instruction, it would be busy and the 146 instruction would have to hold issue.

The vector logical unit performs the logical product (AND), logical sum (OR), and logical difference [XOR (exclusive OR)] functions using either scalar or vector registers.

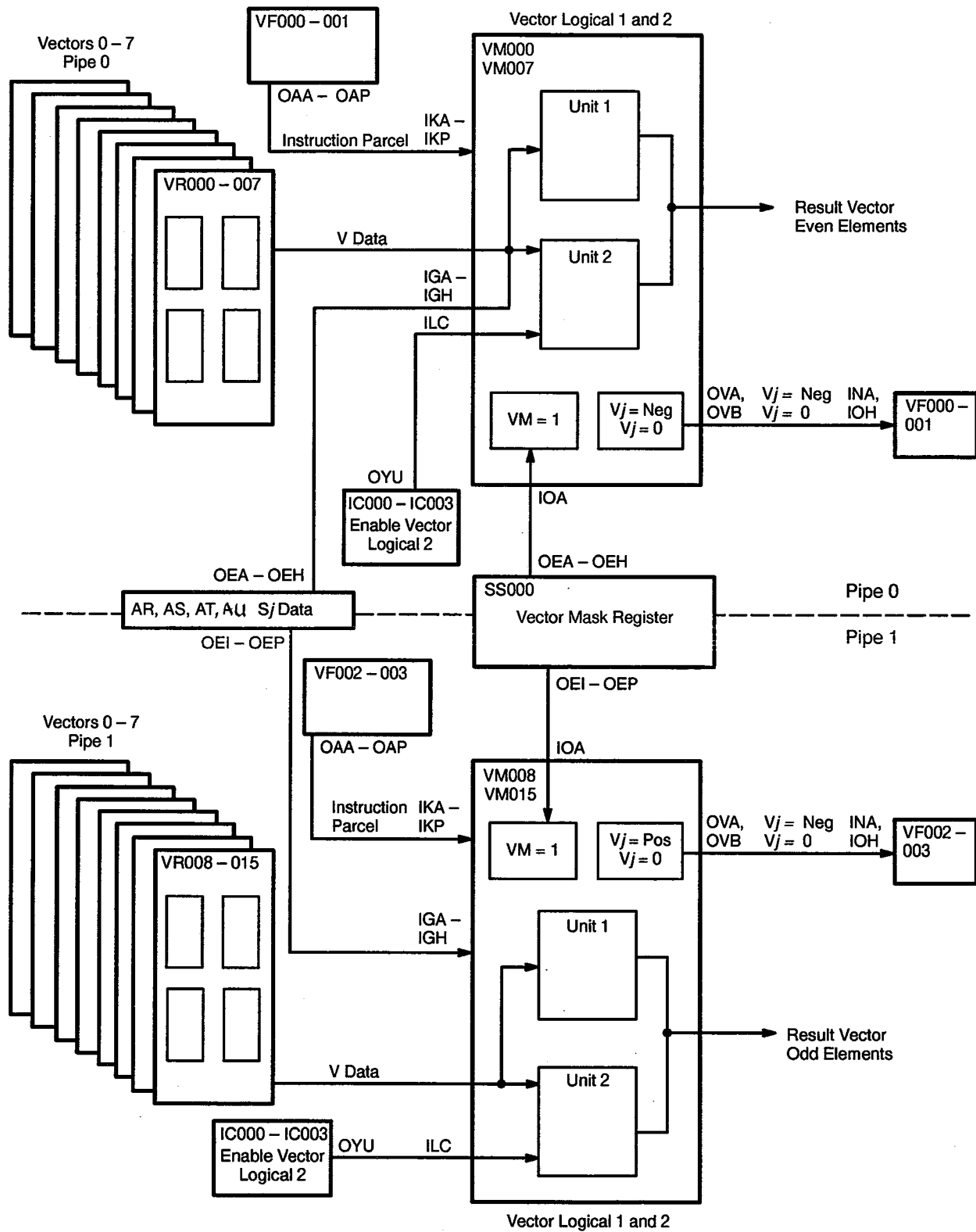


Figure 48. Vector Logical Block Diagram

## Vector Logical Instructions

Refer to Table 15 for a list of the vector logical instructions.

Table 15. Vector Logical Instructions

Instruction	CAL	Description
140ijk	$V_i S_j \& V_k$	Transmit logical product of ( $S_j$ ) and ( $V_k$ elements) to $V_i$ elements
141ijk	$V_i V_j \& V_k$	Transmit logical product of ( $V_j$ elements) and ( $V_k$ elements) to $V_i$ elements
142ijk	$V_i S_j \vee V_k$	Transmit logical sum of ( $S_j$ ) and ( $V_k$ elements) to $V_i$ elements
143ijk	$V_i V_j \vee V_k$	Transmit logical sum of ( $V_j$ elements) and ( $V_k$ elements) to $V_i$ elements
144ijk	$V_i S_j \vee V_k$	Transmit logical differences of ( $S_j$ ) and ( $V_k$ elements) to $V_i$ elements
145ijk	$V_i V_j \vee V_k$	Transmit logical differences of ( $V_j$ elements) and ( $V_k$ elements) to $V_i$ elements

## Vector Merge

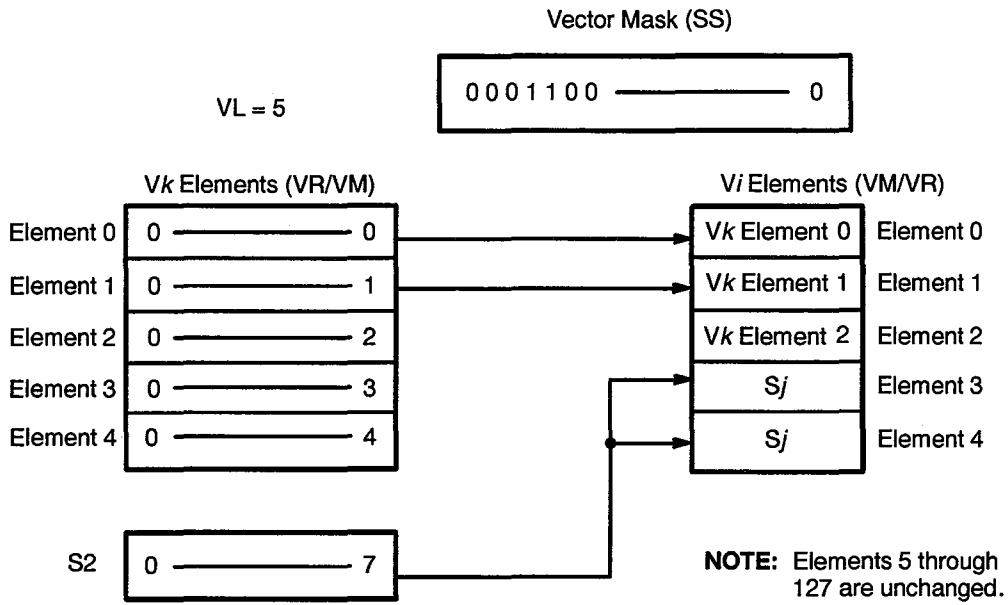
The 146 and 147 instructions merge the contents of the registers using the vector mask register for control. The 146 instruction merges the contents of  $S_j$  with the contents of  $V_k$ ; the 147 instruction merges the contents of  $V_j$  and  $V_k$ . If the vector mask bit is a 1, the  $V_j$  or  $S_j$  data is used; if the vector mask bit is a 0, the  $V_k$  data is used.

The vector logical functional unit holds a copy of the S-register value. Therefore, a subsequent instruction can change the S-register value and not affect the results. These instructions are confined to the second logical unit. Refer to Table 16 for the vector merge instructions, and refer to Figure 49 for an example of a vector merge operation.

Table 16. Vector Merge Instructions

Instruction	CAL	Description
146ijk	$V_i S_j \vee V_k \& VM$	Merge ( $S_j$ ) and ( $V_k$ elements) to $V_i$ elements using ( $VM$ ) as mask
146i0k	$V_i \# VM \& V_k$	Merge 0 and ( $V_k$ elements) to $V_i$ elements using ( $VM$ ) as mask
147ijk	$V_i V_j \vee V_k \& VM$	Merge ( $V_j$ elements) and ( $V_k$ elements) to $V_i$ elements using ( $VM$ ) as mask

147ijk Merge  $S_j$  and  $V_k$  elements to  $V_i$  elements using VM as mask



146ijk Merge  $V_j$  elements and  $V_k$  elements to  $V_i$  elements using VM as mask

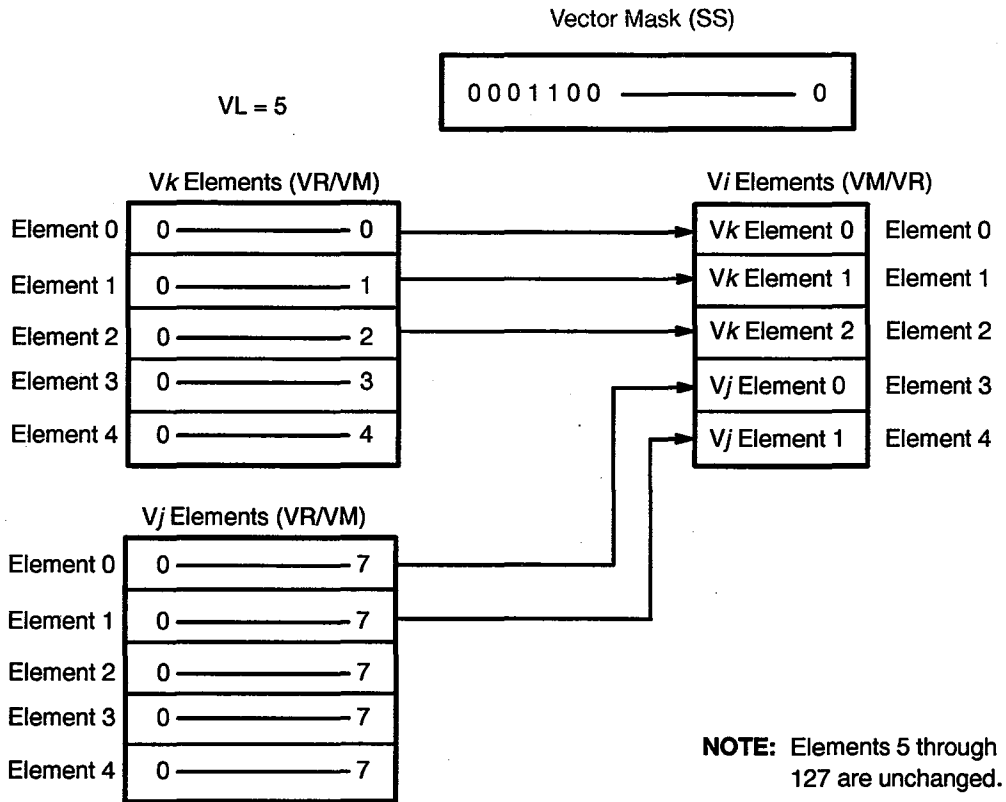


Figure 49. Vector Merge Operation



## Vector Mask

---

There are two vector mask registers: VM0 and VM1. Each register is 64 bits wide, and the two registers are aligned to create a 128-bit register. Each bit in the register corresponds to an element in a vector register. The vector mask register stores the results of a test condition of an element in a vector. For example, a bit can be set in the mask register for all elements in the test vector that are positive values.

The vector mask register receives data from the scalar registers or from the result of comparing a condition within the elements of a vector. The vector mask register is arranged so that mask bit 127 corresponds to element 0 of the vector.

Refer to Table 17 and Table 18 for a list of the vector mask and vector mask test operations, respectively. Refer also to Figure 50 for an illustration of the 1750j0 instructions.

Table 17. Vector Mask Operations

Instruction	CAL	Description
0030j0	VM0 Sj	Transmit (Sj) to VM0
0030j1	VM1 Sj	Transmit (Sj) to VM1
*0030j2	VM0 Aj	Transmit (Aj) to VM0
*0030j3	VM1 Aj	Transmit (Aj) to VM1
070ij1	Vi Ci, Sj & VM	Transmit compressed index of (Sj) controlled by (VM) to Vi
073i00	Si VM0	Transmit (VM0) to Si
073i10	Si VM1	Transmit (VM1) to Si
*073i20	Ai VM0	Transmit (VM0) to Ai
*073i30	Ai VM1	Transmit (VM1) to Ai

\* These instructions must be preceded by a 005400 (EIS) instruction.

Table 18. Vector Mask Test Operations

Instruction	CAL	Description
1750j0	VM Vj,Z	Set VM bit if (Vj element) =0
1750j1	VM Vj,N	Set VM bit if (Vj element) ≠ 0
1750j2	VM Vj,P	Set VM bit if (Vj element) ≥ 0
1750j3	VM Vj,M	Set VM bit if (Vj element) < 0
175ij4	Vi,VM Vj,Z	Set VM bit if (Vj element) = 0 and store compressed indices of Vj elements = 0 in Vi
175ij5	Vi,VM Vj,N	Set VM bit if (Vj element) ≠ 0 and store compressed indices of Vj elements ≠ 0 in Vi
175ij6	Vi,VM Vj,P	Set VM bit if (Vj element) ≥ 0 and store compressed indices of Vj elements ≥ 0 in Vi
175ij7	Vi,VM Vj,M	Set VM bit if (Vj element) < 0 and store compressed indices of Vj elements < 0 in Vi

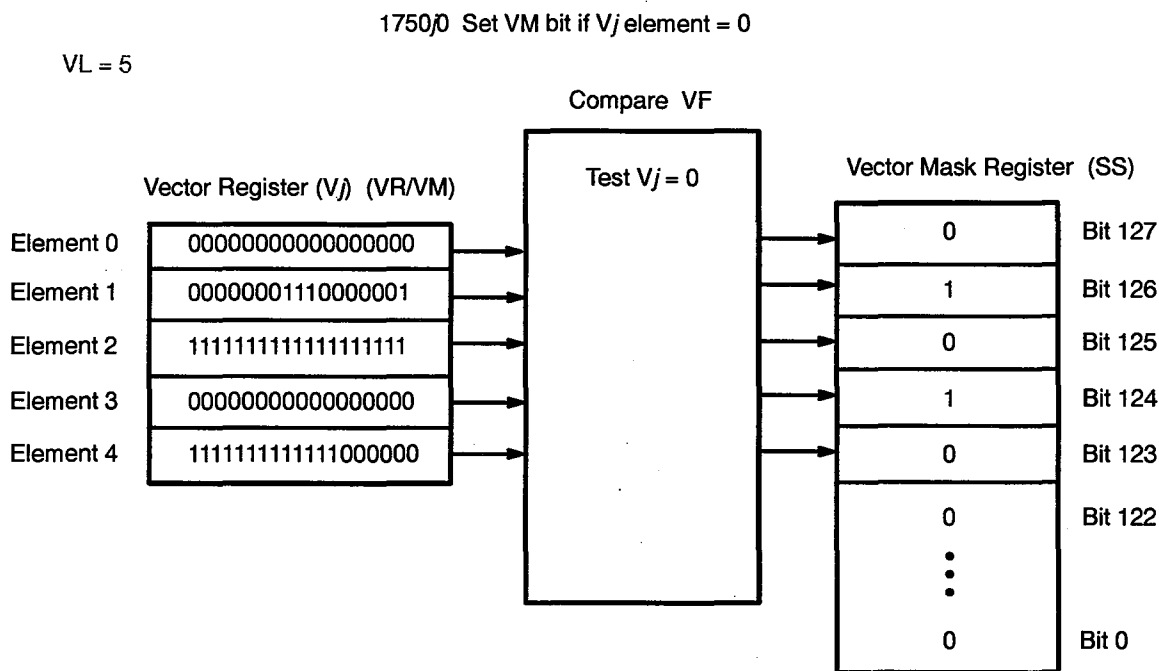


Figure 50. 1750j0 Instructions

Figure 51 illustrates the function of the 175ij4 instructions that use the vector mask to create a compressed vector.

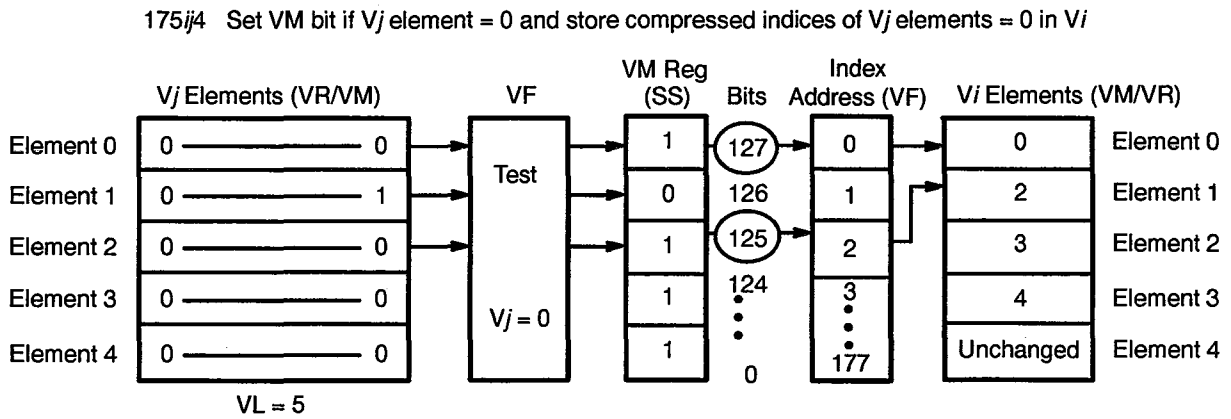


Figure 51. Function of the 175ij4 Instructions

## Compressed Iota

The Iota function is performed on the RA, RB, and RC options; these options also make up the floating-point reciprocal approximation unit and the vector pop functional unit. Table 19 lists the instruction used in iota operations, and Figure 52 is a block diagram of iota pipe 0.

Table 19. Iota Instruction

Instruction	CAL	Description
070ij1	Vi CI,Sj&VM	Transmit compressed index of (Sj) controlled by (VM) to Vi

The 070ij1 instruction forms multiples of the contents of register Sj starting with 0 (0, Sj, 2 x Sj, 3 x Sj, and so on). It stores multiples corresponding to each 1 bit set in the vector mask register in successive elements of register Vi (beginning at element 0). The instruction stops when all unused bits of the vector mask are 0 or are used.

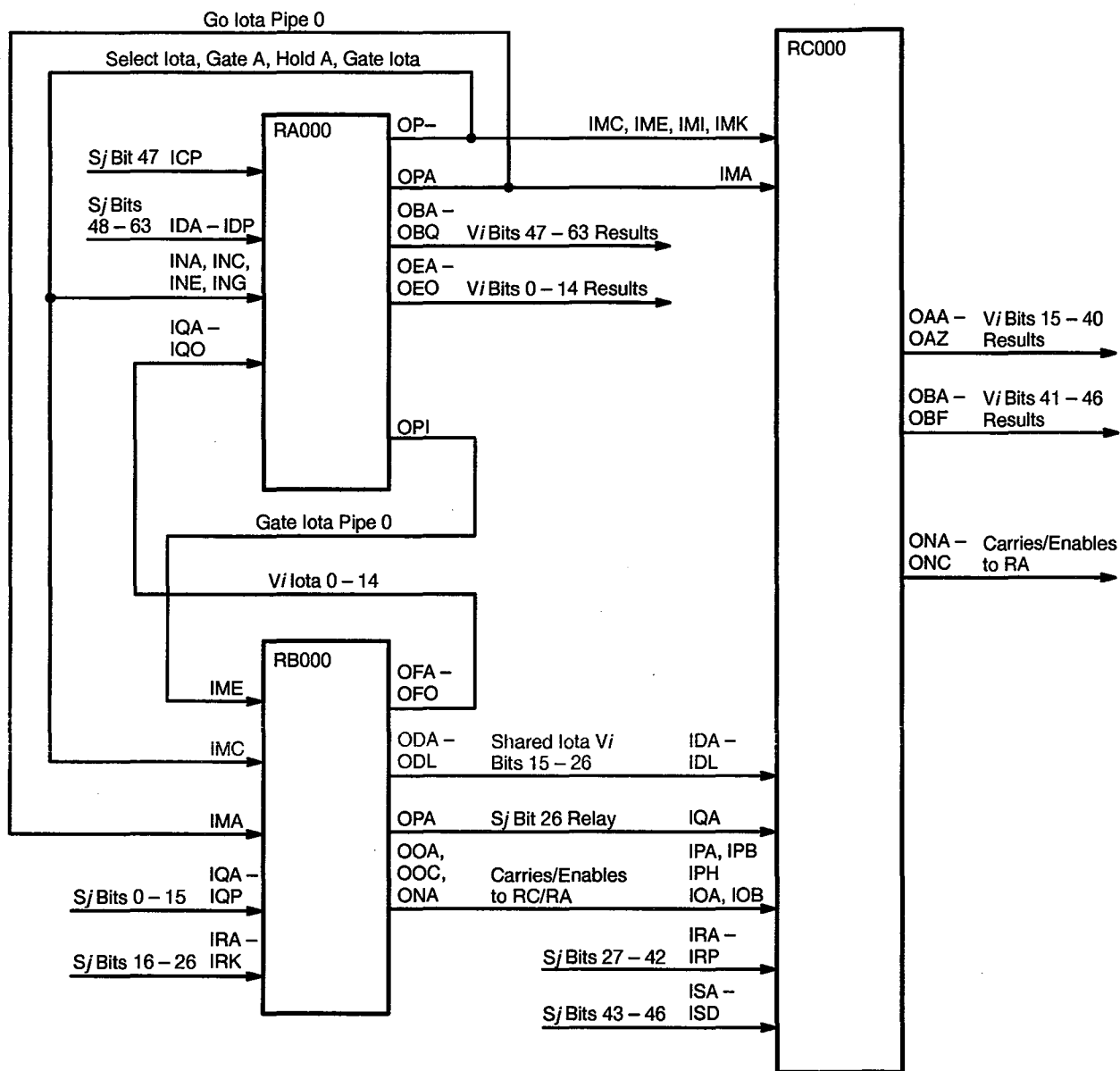


Figure 52. Iota Pipe 0

Figure 53 illustrates the function of the 070ij1 instructions that use the vector mask to create a compressed vector.

070ij1 Transmit compressed index of (Sj) controlled by (VM) to Vi

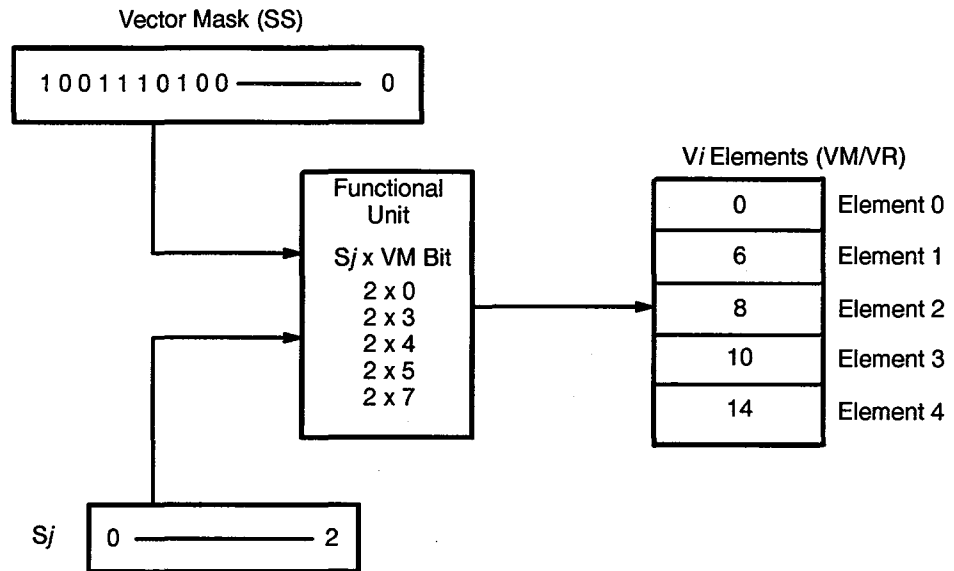


Figure 53. Function of the 070ij1 Instructions

## RA Option

The RA option generates the iota results for bits 47 through 63. It receives iota result bits 0 through 14 from the RB option and outputs bits 0 through 14, and 47 through 63 to the result vector. The RA000 option also generates the control for the iota function for both pipes.

## RB Option

The RB option generates the iota result for bits 0 through 26. Bits 0 through 14 are sent to the RA option, and bits 15 through 26 are sent to the RC option.

The RB option receives two control signals: **Select Iota0** and **Gate Iota**. **Select Iota0** selects the correct iota results from Iota0/Iota1; **Gate Iota** multiplexes (muxes) the iota results to the RA and RC options.

## RC Option

The RC option receives bits 15 through 26 from the RB option and generates result bits 27 through 46 to be sent to the result vectors.

The RC option receives four control signals from the RA option: **Select Iota0**, **Hold A**, **Gate A**, and **Gate Iota**. **Select Iota0** selects from Iota0/Iota1 the correct iota results. **Hold A** and **Gate A** control the first-in-first-out (FIFO) buffers, and **Gate Iota** disables reciprocal/pop/parity/leading zero and enables iota results to be sent to the result vectors.

# VECTOR ADD

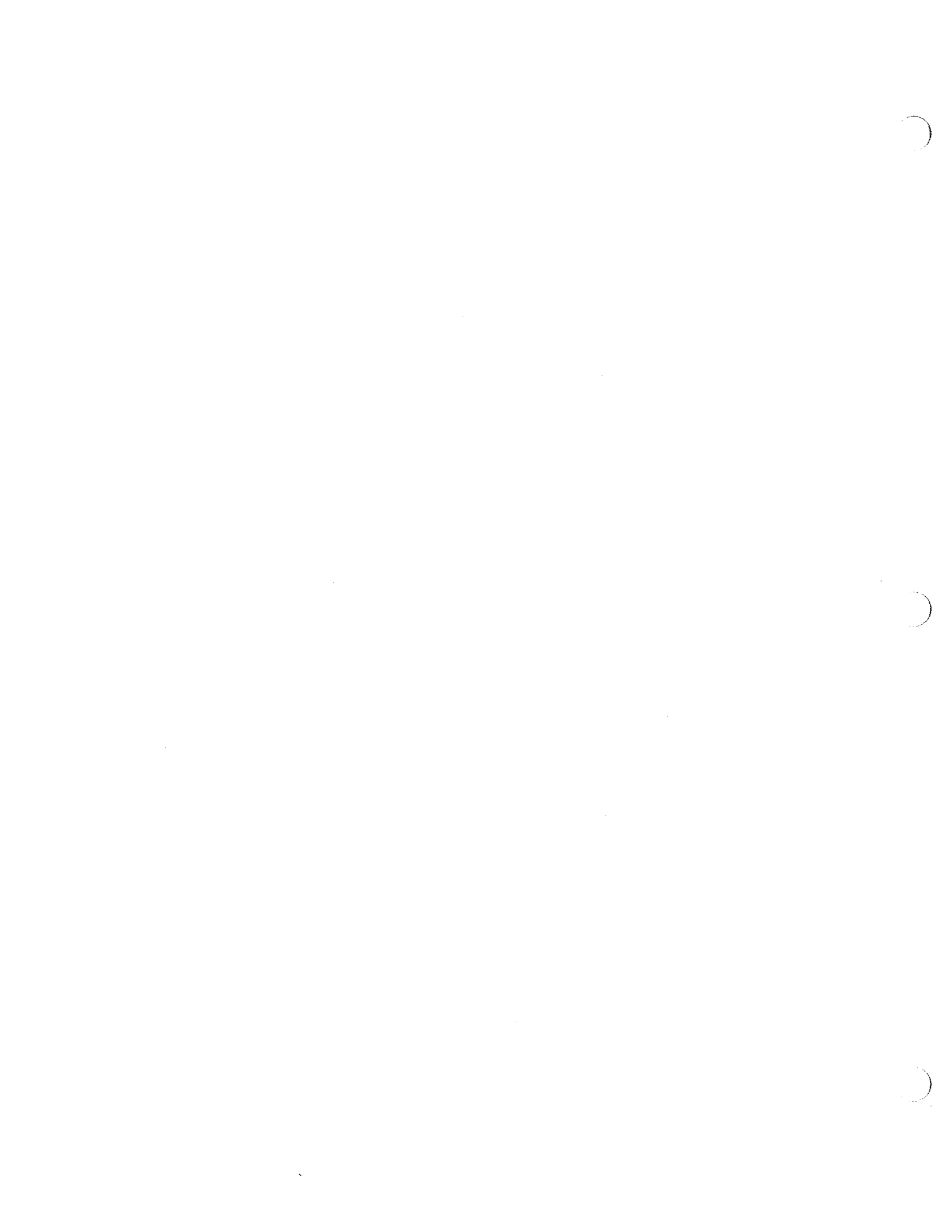
Refer to Figure 54 for a block diagram of vector add. The vector add functional unit is located on the VM and VF options. The VM options perform the actual addition of the input operands and then pass the group carries and group enables to the VF for summation. These bit toggles are then returned to the VM option for final summation. The functional unit uses two's complement arithmetic and does not detect any overflow conditions.

Refer to Table 20 for a list of the vector add instructions and to Figure 54 for a vector add block diagram.

Table 20. Vector Add Instructions

Instruction	CAL	Description
154ijk	$V_i S_j + V_k$	Transmit integer sum of ( $S_j$ ) and ( $V_k$ elements) to $V_i$ elements
155ijk	$V_i V_j + V_k$	Transmit integer sum of ( $V_j$ elements) and ( $V_k$ elements) to $V_i$ elements
156ijk	$V_i S_j - V_k$	Transmit integer difference of ( $S_j$ ) and ( $V_k$ elements) to $V_i$ elements
1560k	$V_i - V_k$	Transmit two's complement of ( $V_k$ elements) to $V_i$ elements
157ijk	$V_i V_j - V_k$	Transmit integer difference of ( $V_j$ elements) and ( $V_k$ elements) to $V_i$ elements

The 154 and 156 instructions use the  $S_j$  register as the second operand. The VM option holds a copy of the S register so if a subsequent instruction wants to use  $S_j$ , that instruction can be changed without affecting the vector instruction.





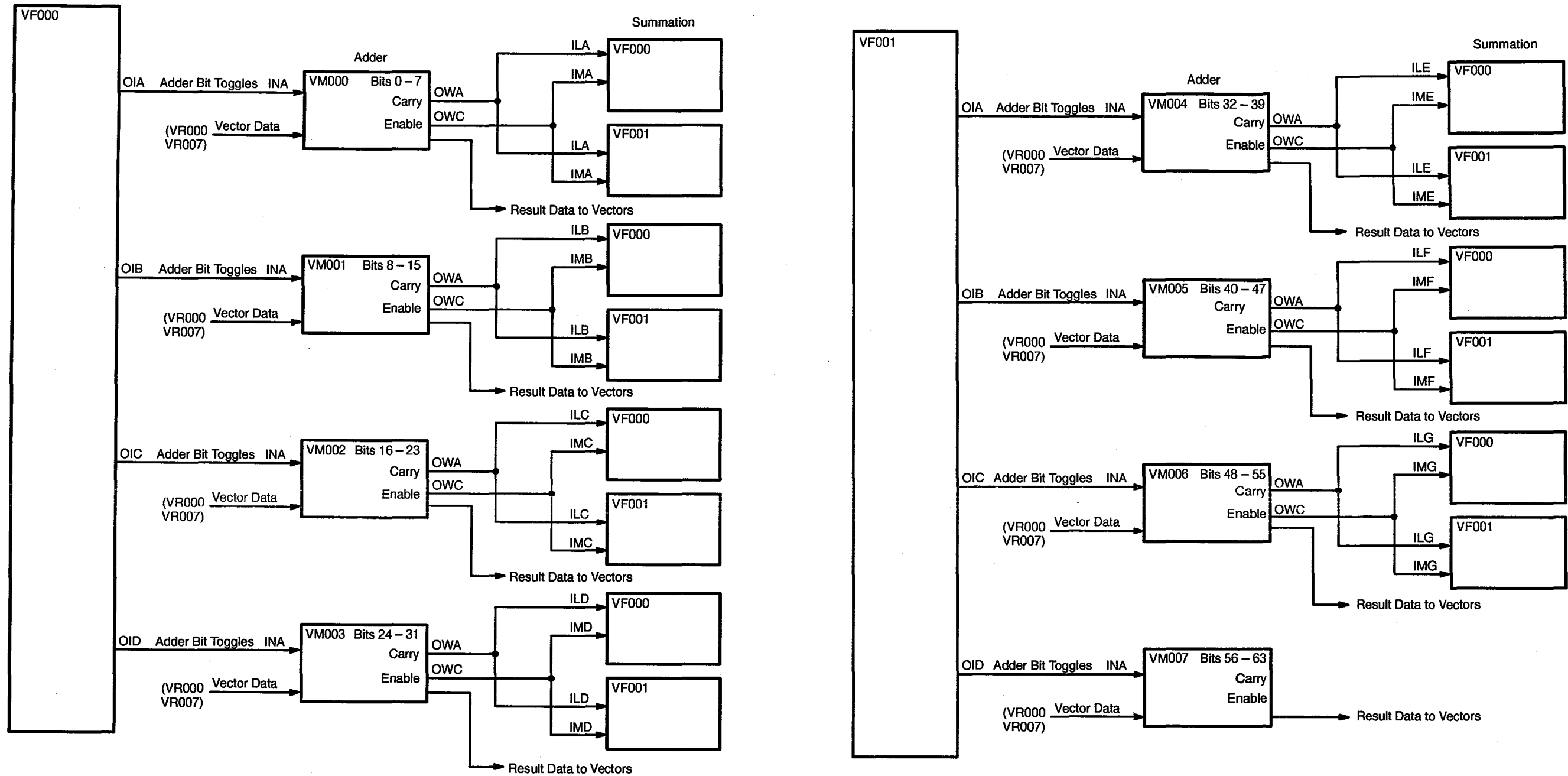


Figure 54. Vector Add Block Diagram

# VECTOR SHIFT

The vector shift functional unit is contained within the VS option. Vector shift is a dual-pipe functional unit; it accepts a pair of elements and generates a pair of results. If the vector length is odd, the last operand generates a single result. There is only one VS option used per CPU.

The vector shift functional unit is also responsible for vector transfer operations. For example, it moves the contents of one vector register to another vector register; then the functional unit uses the  $A_k$  value as a starting element number for the block move.

This unit also performs the vector compress and expand operations. The compress operation writes the elements of  $V_j$  to  $V_i$  if a corresponding bit in the vector mask register sets. The expand operation reads the elements of  $V_j$  to  $V_i$  if a corresponding bit in the vector mask register sets. These operations are illustrated later in this section.

The 150 to 153 instructions use  $A_k$  as the shift count. The 150 to 151 instructions, when preceded by a 005400 (EIS) instruction, use  $V_0$  for the shift count. In either case, if bit 7 or above is set, the result is 0's.

## Vector Shift Instructions

Refer to Table 21 for a list of the vector shift instructions.

Table 21. Vector Shift Instructions

Instruction	CAL	Description
150ijk	$V_i V_j < A_k$	Shift ( $V_j$ elements) left ( $A_k$ places) to $V_i$ elements
*150ij0	$V_i V_j < V_0$	Shift ( $V_j$ elements) left ( $V_0$ elements) places to $V_i$ elements
151ijk	$V_i V_j > A_k$	Shift ( $V_j$ elements) right ( $A_k$ places) to $V_i$ elements
*151ij0	$V_i V_j > V_0$	Shift ( $V_j$ elements) right ( $V_0$ elements) places to $V_i$ elements
152ijk	$V_i V_j, V_j < A_k$	Double shift ( $V_j$ elements) left ( $A_k$ places) to $V_i$ elements
*152ijk	$V_i V_j, A_k$	Transfer ( $V_j$ elements) starting at element ( $A_k$ ) to $V_i$ elements
153ijk	$V_i V_j, V_j > A_k$	Double shift ( $V_j$ elements) right ( $A_k$ places) to $V_i$ elements

\* These instructions must be preceded by a 005400 (EIS) instruction.

Table 21. Vector Shift Instructions (continued)

Instruction	CAL	Description
*153j0	$V_i V_j \{VM\}$	Compress $V_j$ by (VM) to $V_i$
*153j1	$V_i [VM] V_j$	Expand $V_j$ by (VM) to $V_i$

\* These instructions must be preceded by a 005400 (EIS) instruction.

## Vector Shift Count Description

The  $A_k$  shift count is sent to the VS option by the AR000 option, and all eight A series options check the value of the 64-bit A register. This determines if any bits above bit 6 have been set. If any bits have been set, the result is lost due to overshift. If no overflow is detected, a **No  $A_k$  Overflow** signal is sent from the SS to the VS. AR000 sends bits 0 through 6 for the shift count.

To understand this, the breakdown of the shift count must be examined. For both single and double shifts, the breakdown is the same, except for the fact that the double shift has 1 extra bit (bit 6). Refer to Figure 55 for a breakdown of the shift count, and to Figure 56 for a block diagram of vector shift.

Double Shift Only								
6	5	4	3	2	1	0	Bit Position	
64	32	16	8	4	2	1	Shift Value	

Figure 55. Shift Count Breakdown

Each bit position of the shift count represents a shift value, and the sum of the shift value for each bit set in the shift count equals the total number of places shifted. The maximum shift count that could be generated is  $127_{10}$  or  $177_8$ .

**NOTE:** The shift value is shown as a decimal value; all references to shift counts in the documentation refer to a decimal count. Also, a shift of 0 generates a maximum shift of  $177_8$  places; this zeroes out the result register.

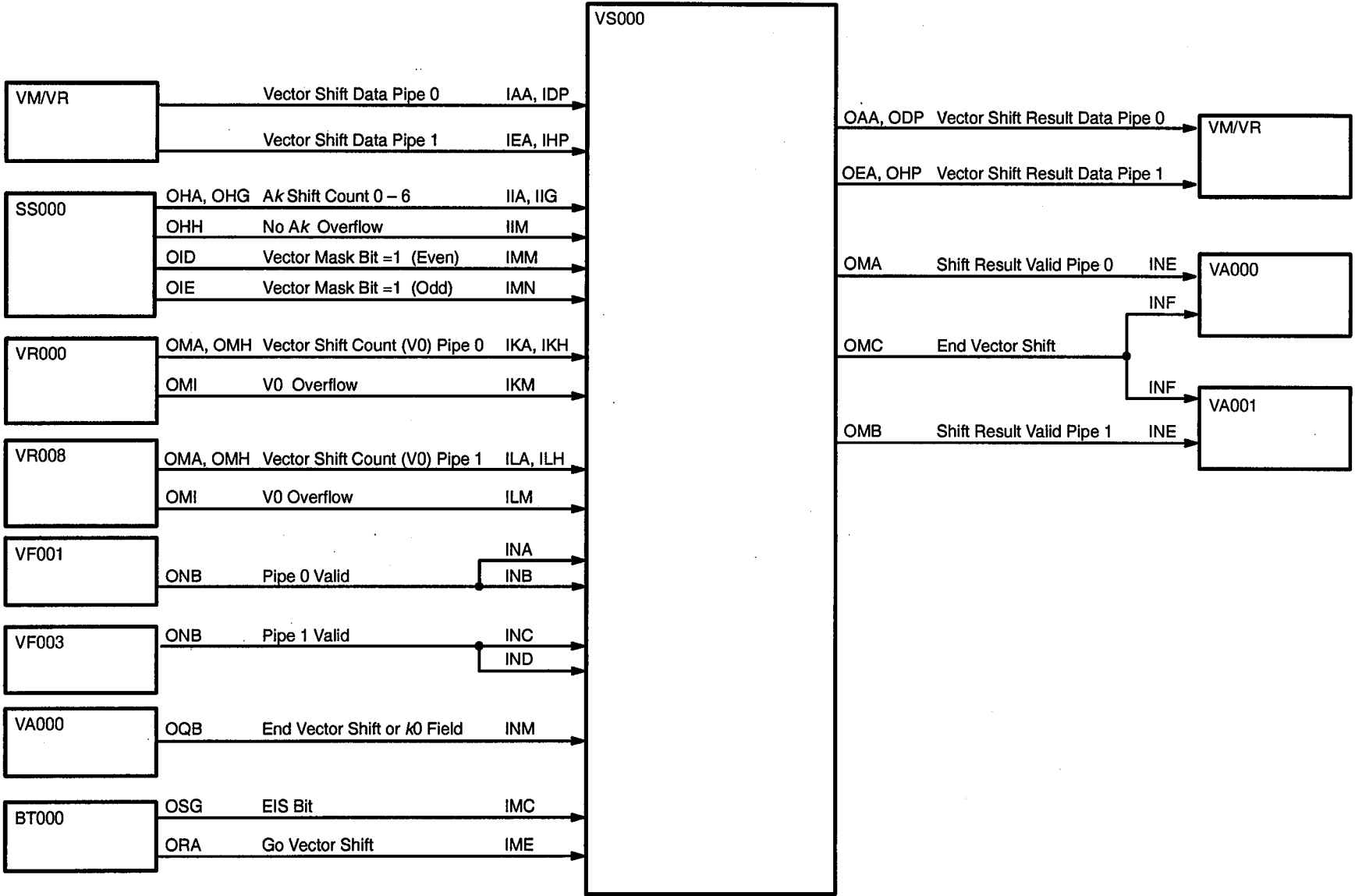


Figure 56. Vector Shift Block Diagram

If the *jk* field of a left single shift equals  $27_8$  and bits 4, 2, 1, and 0 are set, the shift values are 16, 4, 2, and 1, respectively. The sum of the shift values is 23 ( $16 + 4 + 2 + 1$ ); therefore, the instruction shifts left  $23_{10}$  places.

The actual hardware that performs the shifts is the same for both left and right shifts. However, the hardware performs only left shifts. Right shifts are accomplished according to the way data is entered into the shifter, hence the use of two's complement for right shifts.

The vector shift unit also receives a shift count from V0 when performing the 150 and 151 EIS instructions. The shift count is sent to the VS option from VR0 for pipe 0 and from VR8 for pipe 1.

### Vector Right Shift 005400 151 *ij*0

Refer to Figure 57 for an example of a vector right shift using V0 for the shift count. Note that the shift count for element 0 is 0; this results in an end-off shift for that element. This instruction must be preceded by the 054100 instruction in order to function as illustrated. This process continues for vector length.

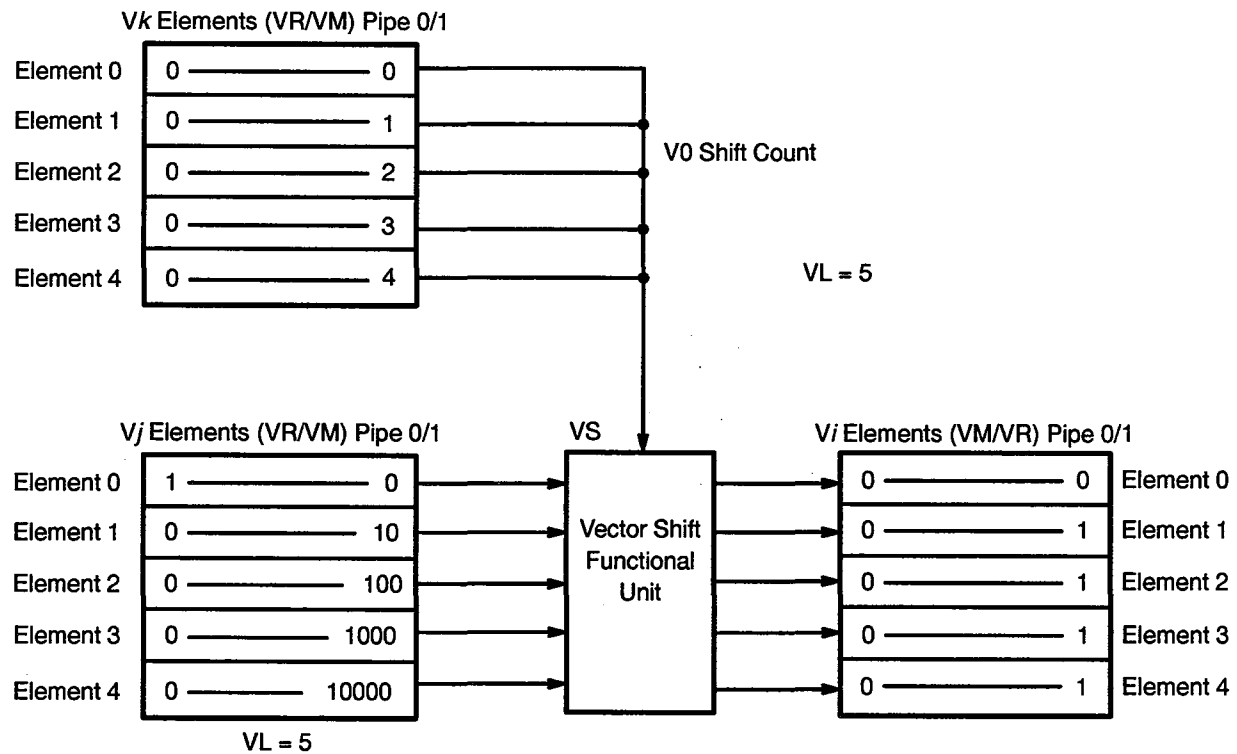


Figure 57. Vector Right Shift

### Vector Right Double Shift 153ijk

Refer to Figure 58 for an example of a vector right double shift using  $A_k$  for the shift count. This instruction concatenates two successive elements of register  $V_j$  and right shifts the lower 64 bits to  $V_i$ . The first operation combines element 0 with a word of all 0's. Element 0 becomes the lower 64 bits, and this value is then shifted right  $A_k$  places to  $V_i$ .

The next operation combines element 0 and element 1 of  $V_j$ , with element 1 being the least significant bits, and shifts this value right to  $V_i$ . This operation continues for vector length. Note that the shift count for element 0 is 0; this results in an end-off shift for that element.

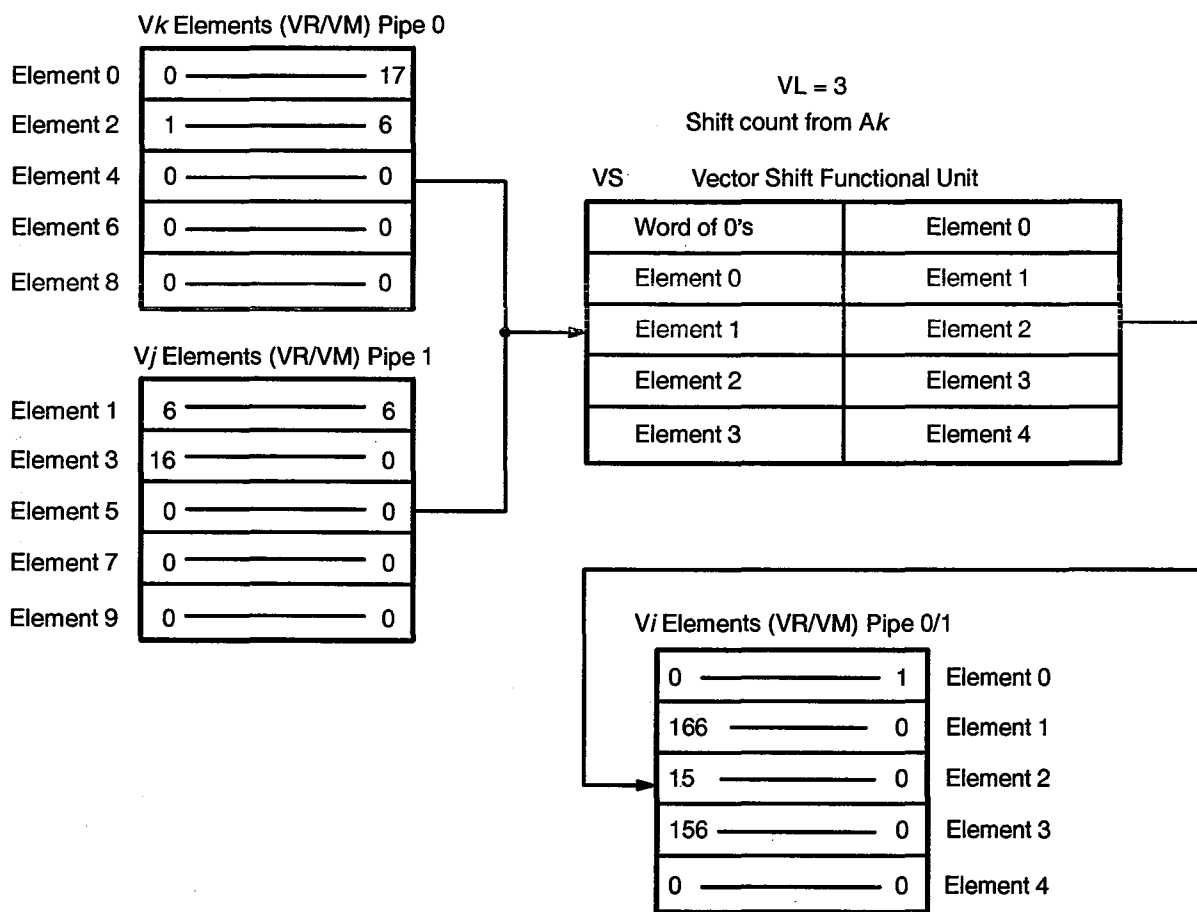


Figure 58. Vector Right Double Shift

### Vector Transfer 005400 152ijk

This instruction moves the contents of  $V_j$  to  $V_i$  starting with element  $A_k$  as shown in Figure 59. Note that this is an EIS instruction.

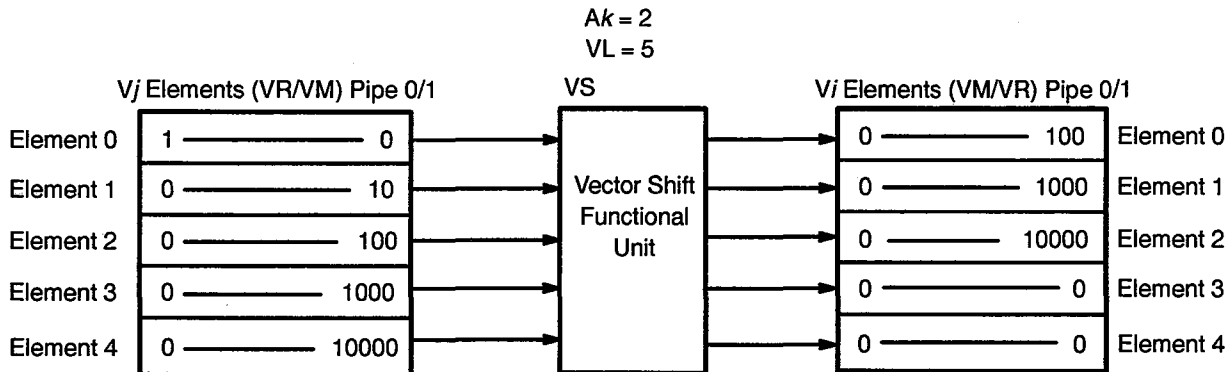


Figure 59. Vector Transfer

### Vector Compress 005400 153ij0

This instruction compresses a vector register using a vector mask and transmits the results to  $V_i$  as shown in Figure 60.

Two element counters are initialized to 0, one for  $V_j$  and the other for  $V_i$ . The vector mask is then scanned from right to left, and for every 1 bit set, an element of  $V_j$  is written to  $V_i$ . The element counters internal to the VS option determine the element position within each register.

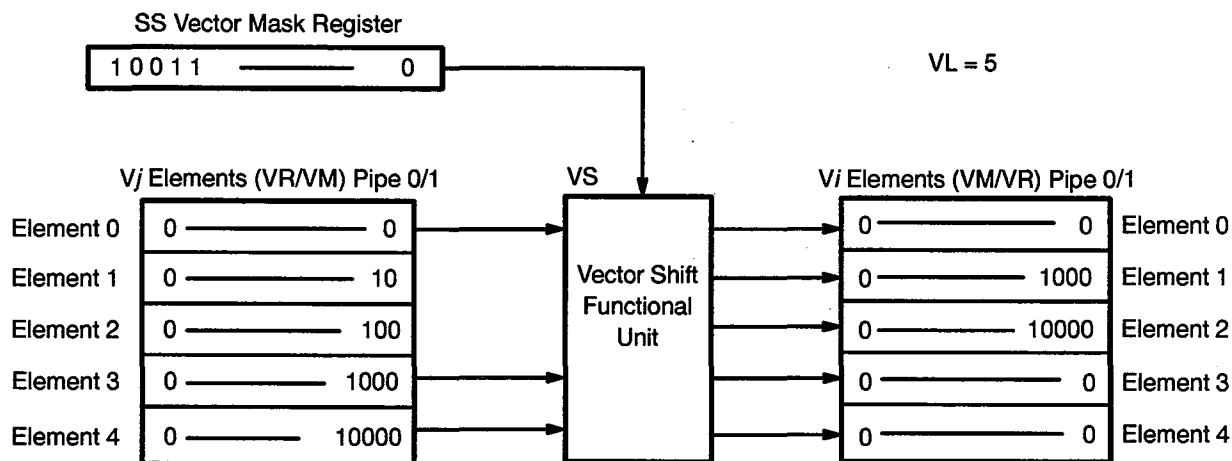


Figure 60. Vector Compress

## Vector Expand 005400 153ij1

This instruction expands a vector register using a vector mask and transmits the results to  $V_i$  as shown in Figure 61.

Two element counters are initialized to 0, one for  $V_j$  and the other for  $V_i$ . The vector mask is then scanned from right to left, and for every 1 bit set, an element of  $V_j$  is written to  $V_i$ . The element counters internal to the VS option determine the element position within each register. In this instruction, the element counter for  $V_j$  falls behind the counter for  $V_i$  by one position for each 0 bit in the vector mask register.

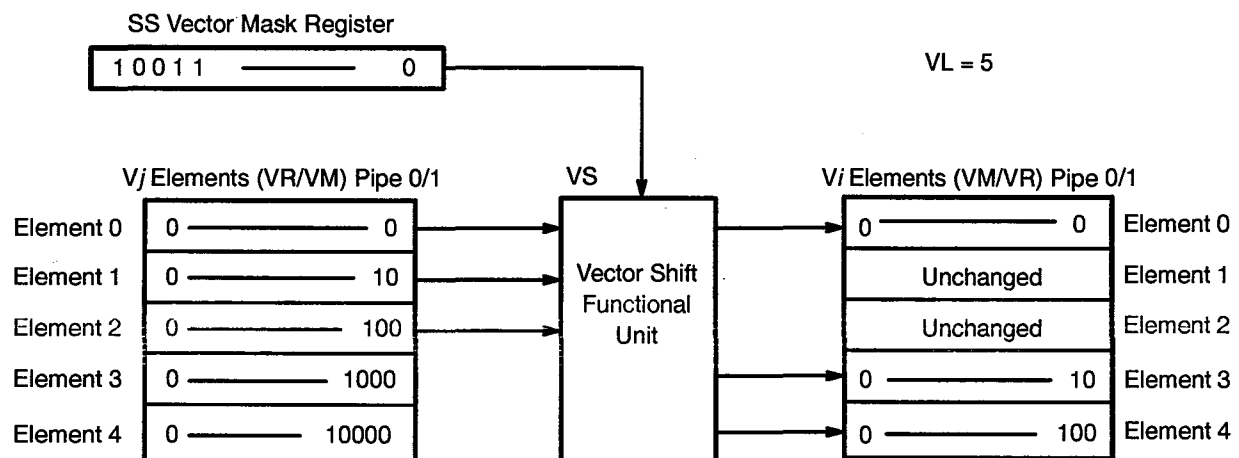
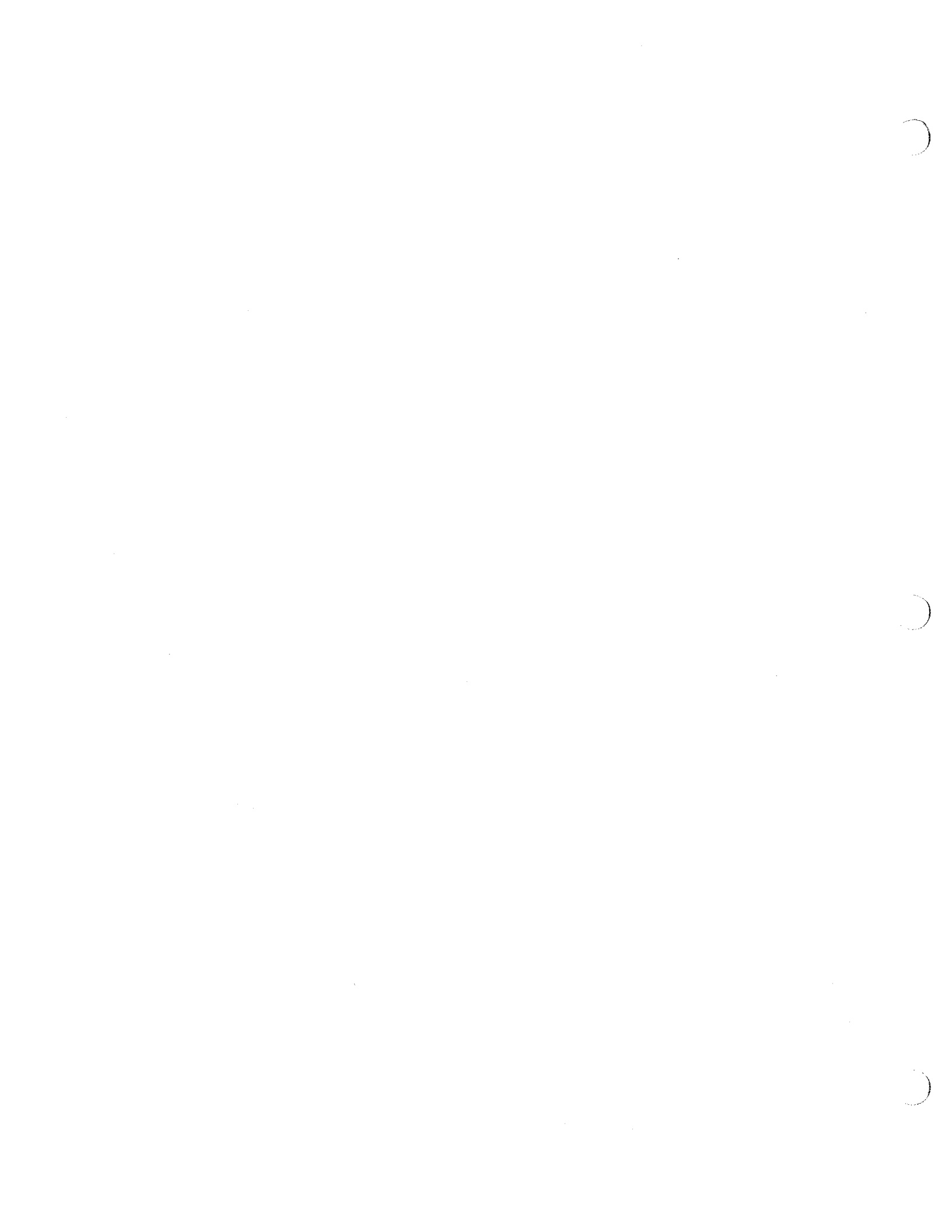


Figure 61. Vector Expand





# VECTOR POP/ POP PARITY AND LEADING ZERO

The vector population/parity functional unit performs population counts and parity for vector operations and executes instructions 174ij1 vector population count and 174ij2 vector parity.

Refer to Figure 62 for a vector population/parity/leading zero block diagram. This functional unit shares logic with the floating-point reciprocal approximation functional unit. The  $k$  field of the instruction determines the type of operation to be performed.

Because the vector population/parity functional unit shares logic with the floating-point reciprocal approximation functional unit, all vector operations reserve the associated functional unit. The floating-point reciprocal approximation functional unit is reserved when the vector population/parity functional unit is reserved and vice versa.

Both scalar and vector register operations share the floating-point reciprocal functional unit. Therefore, when vector reciprocal or vector population/parity instructions are executed, any scalar reciprocal instruction holds issue until the vector operation is finished.

The 174ij1 instruction counts the number of 1 bits in each element of a vector register specified by  $V_i$ . Each element is counted individually, and the result is stored in the respective element of  $V_i$ . For example, the count of 1 bits in element 0 of  $V_j$  is stored in element 0 of  $V_i$ ; the count of 1 bits in element 1 of  $V_j$  is stored in element 1 of  $V_i$ ; and so on. This process continues for the number of elements equal to the VL.

The 174ij2 instruction counts the number of 1 bits in each element of a vector register specified by  $V_j$  and stores a 1-bit parity result in a vector register specified by  $V_i$ . The 174ij2 instruction uses the same logic as the 174ij1 but outputs only bit 0 of the result. Bits 1 through 6 are forced to 0's. This instruction determines whether an odd or even number of bits are set in each element of a vector register. If the result equals 0, there is an even number of bits. If the result equals 1, there is an odd number of bits.

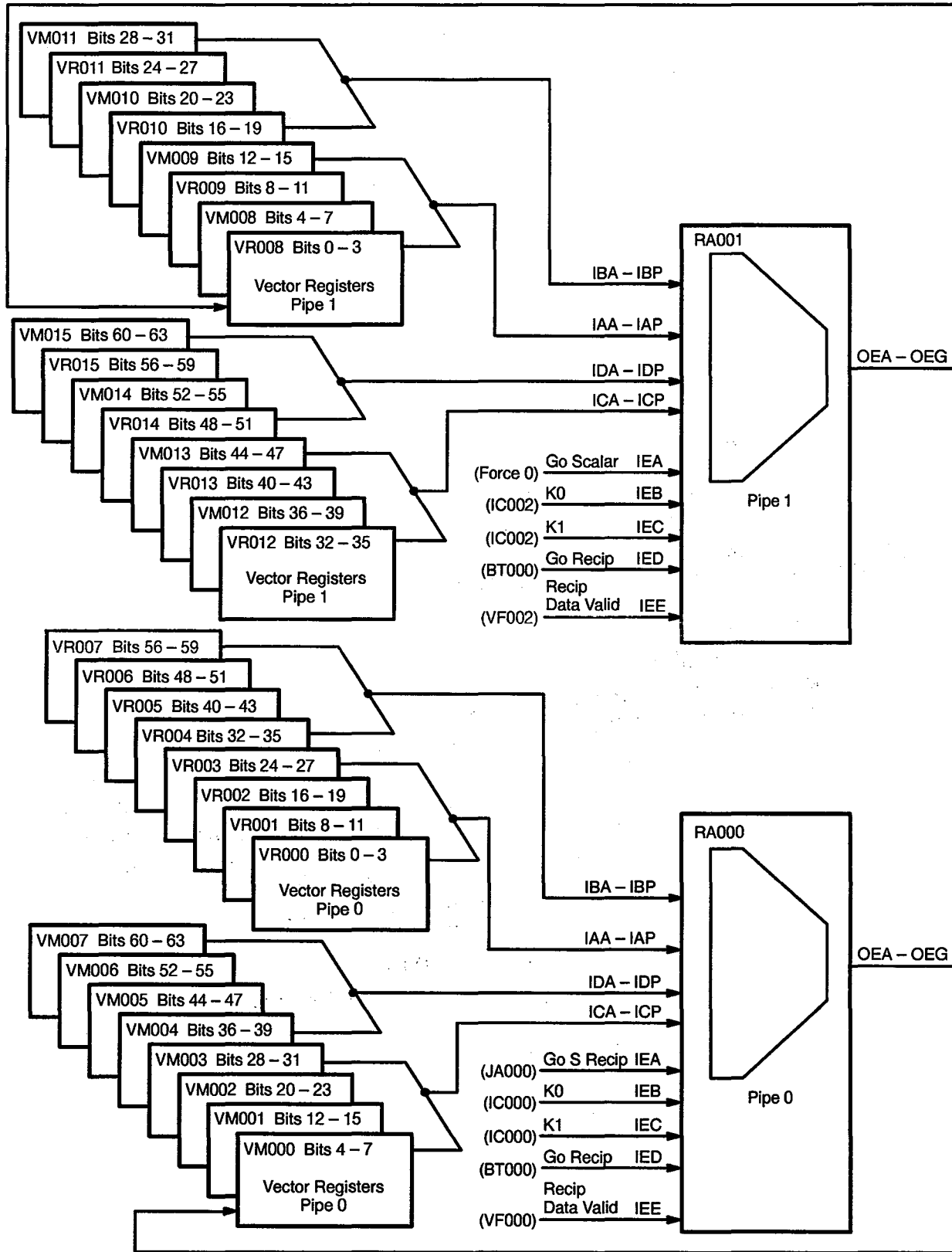


Figure 62. Vector Population/Parity/Leading Zero Block Diagram

## **Pop/Parity/Leading Zero Functional Units**

---

The RA options contain part of the reciprocal approximation unit; these options also contain the logic for vector pop, vector pop parity, and vector leading zero. There are two RA options per CPU: RA000 handles pipe 0, or the even elements; and RA001 handles pipe 1, or the odd elements.

The RA options receive data from the VM and VR options; 4 bits come from each VR and VM. Data is sent on the same wires and terms that the reciprocal data uses. The data is then sent to VM000 and VM008 on the same terms that the reciprocal output data uses. Data is sent to only those two options because the pop functional unit returns only a 7-bit value to the result register.

### **Vector Population Count 174ij1**

Vector pop counts the number of bits set in an element and reports that count to a result vector. The count ranges anywhere from 0 (no bits in the element set) to 100 (all bits in the element set). The functional unit sends only bits 0 through 6 to the result vector; the remaining bits are zeroed out.

### **Vector Population/Parity 174ij2**

This instruction counts the number of bits set in each element of a vector and then determines whether this number of bits is an even or an odd number. If the result is an even number of bits, a 0 is written to the result vector. If the number of bits is odd, a 1 is written to the result vector. Only bit 0 is written to the result vector; the rest of the bits in the element are set to 0's.

### **Vector Leading Zero Count 174ij3**

This instruction counts the number of 0's that precede the first bit set in each element of a vector. The count will be from 0 (bit 63 of the element set) to 100 (no bits in the element set).

## Vector Population/Parity Instructions

---

Refer to Table 22 for a list of the vector population/parity instructions.

Table 22. Vector Population/Parity Instructions

Instruction	CAL	Description
174ij1	$V_i PV_j$	Population count ( $V_j$ ) to $V_i$
174ij2	$V_i QV_j$	Parity of ( $V_j$ ) to $V_i$
175ij3	$V_i ZV_j$	Transmit leading zero count of ( $V_j$ ) to $V_i$

# GATHER/SCATTER INSTRUCTIONS

The *176i1k* and *1771jk* instructions transfer blocks of data between common memory and the vector registers. The *176* invokes the gather, or read function; the *177* invokes the scatter, or write function. When the *176i1k* instruction is preceded by a *005400* instruction parcel, it performs a double gather function, which utilizes the dual-pipe capability of the computer system. The contents of the vector length (VL) register determine the number of words transferred.

## Gather Instructions

---

The *176i1k* instruction transfers data from common memory to the  $V_i$  register. Register  $A_0$  contains the initial (base) address; the  $V_k$  register contains the address indices.

For each element transferred to  $V_i$ , the memory address is the sum of  $(A_0)$  and the corresponding element of register  $V_k$ . For example, during a *176213* instruction,  $V_2[0]$  is loaded from address  $(A_0) + (V_3[0])$ ;  $V_2[1]$  is loaded from address  $(A_0) + (V_3[1])$ ; etc.

The *005400 176ijk* instruction performs the double gather operation. Data is transferred from common memory to  $V_i$  and  $V_j$  in two separate data transfers that occur simultaneously. The  $A_0$  register contains the base address for the transfer to  $V_i$ . The  $A_k$  register contains the base address for the transfer to  $V_j$ . The  $V_k$  register contains the address indices for both transfers.

For each element transferred to  $V_i$ , the memory address is the sum of  $(A_0)$  and the corresponding element of  $V_k$ . For example, during a *005400 176213* instruction,  $V_2[0]$  is loaded from address  $(A_0) + (V_3[0])$ ;  $V_2[1]$  is loaded from address  $(A_0) + (V_3[1])$ ; etc. Simultaneously,  $V_1[0]$  is loaded from address  $(A_3) + (V_3[0])$ ;  $V_1[1]$  is loaded from address  $(A_3) + (V_3[1])$ ; etc.

## Scatter Instructions

---

The  $1771jk$  instruction transfers data from  $Vj$  to common memory. The  $A0$  register contains the initial address.  $Vk$  contains the address indices.

For each element transferred from register  $Vi$ , the memory address is the sum of  $(A0)$  and the corresponding element of register  $Vk$ . For example, element 0 of  $Vi$  is stored to address  $(A0) + (Vk[0])$ ; element 1 of  $Vi$  is stored to address  $(A0) + (Vk[1])$ ; etc.

# FLOATING-POINT ADD

Refer to Figure 63 for a block diagram of floating-point add. The floating-point add unit consists of two option types: the FA and the FB options. Each pipe has one FA option and one FB option. FA000 and FB000 represent pipe 0, and FA001 and FB001 represent pipe 1. The use of dual pipes allows two floating-point add functions to occur at the same time. The even elements of the vector go to pipe 0; the odd elements go to pipe 1. This feature helps in troubleshooting; if you identify which element is failing, you can identify which pipe and associated options are failing. For scalar floating-point add instructions, only pipe 0 is used.

The floating-point add unit must do several things to produce a result. First, the exponents of the input operands must be compared to determine which is larger. Then, the coefficient of the smaller must be right shifted until the exponents become equal. When this is done, the coefficient is then added. If the sign bits are different, or if the sign bits are the same and a subtract instruction is decoded, then a two's complement addition is performed.

Next, the results have to be normalized and the exponent adjusted. The results are then sent to the result registers (either scalar or vector registers). Finally, if the resulting exponent is greater than  $60000_8$  or less than  $17777_8$ , the results are checked for overflow and underflow conditions. If an overflow condition exists, the exponent is forced to  $60000_8$ , the coefficient is left intact, and an error flag is set in the exchange package. If an underflow condition exists, the exponent and the coefficient are forced to 0 and no flag is set. The result coefficient is also checked for a zero value. If it is 0, both the result exponent and coefficient are zeroed out.

The issuing of a 005400 extended instruction set (EIS) instruction just before a floating-point add instruction enables the extended accuracy mode. This adds a rounding bit if all the necessary conditions are satisfied. This is accomplished with the use of *sticky bits*. When the operand of the smaller exponent number is right shifted to equalize the exponents, the coefficient may be shifted more than  $47_8$  places, resulting in a coefficient of 0. What actually takes place is the bits are shifted right into another register as bit -1 to -15, as shown in Figure 64. If any of these bits set and EIS sets, a rounding bit is added to the result coefficient at bit position 0.



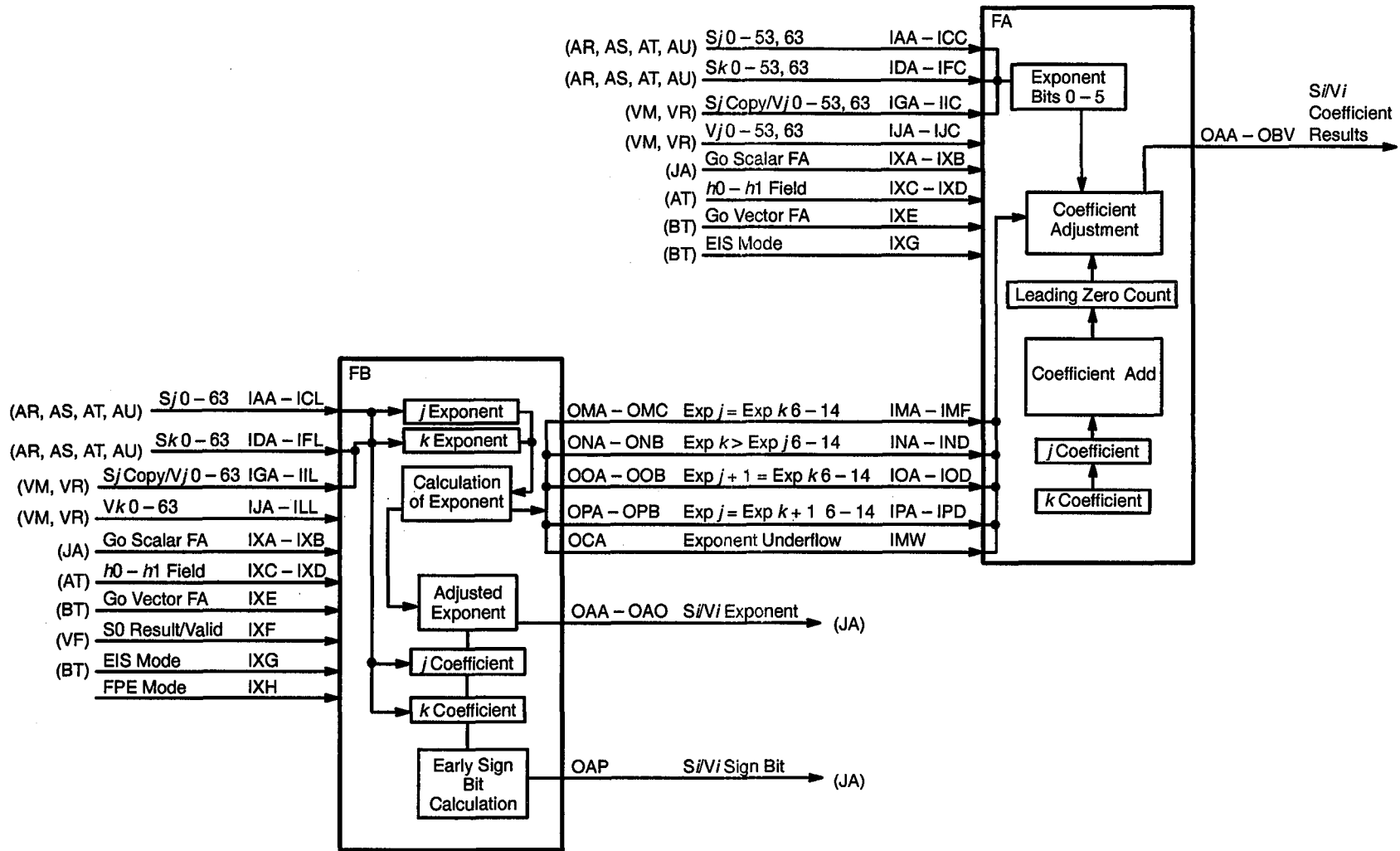


Figure 63. Floating-point Add



Figure 64. Floating-point Add Sticky Bits

## Floating-point Add Functional Unit Instructions

Refer to Table 23 for a list of the floating-point add functional unit instructions.

Table 23. Floating-point Add Functional Unit Instructions

Instruction	CAL	Description
062ijk Si	$S_j + FSk$	Scalar floating-point sum of (Sj) and (Sk) to Si
062i0k	$S_i + FSk$	Transmit normalized (Sk) to Si
063ijk	$S_i S_j - FSk$	Scalar floating-point difference of (Sj) minus (Sk) to Si
063i0k	$S_i - FSk$	Transmit normalized negative of (Sk) to Si, normalize the coefficient and toggle the sign bit
170ijk	$V_i S_j + FV_k$	Vector floating-point sum of (Sj) and (Vk elements) to Vi
171ijk	$V_i V_j + FV_k$	Vector floating-point sum of (Vi elements) and (Vk elements) to Vi
172ijk	$V_i S_j - FV_k$	Transmit normalized negatives of (Vk elements) to Vi, normalize the coefficient and toggle the sign bit
173ijk	$V_i V_j - FV_k$	Vector floating-point difference of (Vj elements) minus (Vk elements) to Vi

## Floating-point Format

Refer to Figure 65 for an illustration of floating-point format. A number is referred to as *normalized* if the upper bit of the coefficient (bit 47) is set.



Figure 65. Floating-point Format

## Floating-point Add Examples

---

Refer to the following subsections for some examples of floating-point add.

### Add Instruction (Subtract Operation)

$$\begin{array}{r}
 j = 040002 \ 140000 \ 000000 \ 000000 + 3_8 \\
 k = 140003 \ 140000 \ 000000 \ 000000 + \frac{-6_8}{-3_8}
 \end{array}$$

#### Subtract Operation

Shift <i>j</i>	040003	060000	000000	000000
Retain <i>k</i>	040003	060000	000000	000000
Toggle <i>k</i>	<u>140003</u>	<u>037777</u>	<u>177777</u>	<u>177777</u>
<u>Add</u> coefficients	140003	117777	177777	177777

CBP (carry across binary point)

Retain exponent and sign of larger

Toggle result	140003	0600000	00000	000000
Normalize	140002	140000	000000	000000

**Subtract Instruction (Add Operation)**

$$\begin{array}{r}
 j = 040003 \ 140000 \ 000000 \ 000000 \quad 6_8 \\
 k = 140002 \ 140000 \ 000000 \ 000000 \quad -3_8 \\
 \hline
 \quad \quad \quad \quad \quad \quad \quad \quad \quad 11_8
 \end{array}$$

**Add Operation**

J operand	040003	140000	000000	000000
Complement $k$ sign bit	040002	140000	000000	000000
Retain $j$	040003	140000	000000	000000
Shift $k$	040003	060000	000000	000000
<u>Add</u> coefficients	040003	1.020000	000000	000000
CBP				
	040004	110000	000000	000000

Shift right to normalize; adjust exponents

**Add Instruction (Subtract Operation with Carry across Binary Point)**

$$\begin{array}{r}
 j = 040004 \ 004000 \ 000000 \ 000000 \quad .4_8 \\
 k = 140003 \ 140000 \ 000000 \ 000000 \quad + -6.0_8 \\
 \hline
 \quad \quad \quad \quad \quad \quad \quad \quad \quad -5.4_8
 \end{array}$$

**Subtract Operation**

Retain $j$	040004	004000	000000	000000
Shift $k$	140004	060000	000000	000000
Toggle $j$	040004	173777	177777	177777
	140004	060000	000000	000000
<u>Add</u> coefficients	040004	1.053777	177777	177777
CBP				

Retain exponent and sign of larger

040004 053777 177777 177777

+1 End-around carry

Toggle sign bit 140004 054000 000000 000000

Normalize 140003 130000 000000 000000

**Add Instruction (Add Operation)**

$$\begin{array}{r}
 j = 040003 \ 140000 \ 000000 \ 000000 \ 6_8 \\
 k = 040002 \ 140000 \ 000000 \ 000000 \ + \ 3_8 \\
 \hline
 11_8
 \end{array}$$

Add Operation

Retain *j* 040003 140000 000000 000000

Shift *k* 040003 060000 000000 000000

Add coefficients 040003 1.020000 000000 000000

040004 110000 000000 000000

CBP

Normalize result

**FA Option**

---

The FA option operates on the coefficient portion of the floating-point add operation. The FA does the actual addition of the *j* and *k* operands. It also determines from the sign bit and the instruction issued whether to perform an add or subtract operation.

If the extended accuracy mode is set by an EIS instruction, a rounding bit is inserted into the result coefficient if all the necessary conditions are satisfied.

The FA option also uses the lower 6 bits of the exponent (48 through 53) and control signals sent from the FB option to make the final determination of the right shift, which aligns the coefficient.

## FB Option

---

The FB option operates on the exponent portion of the floating-point add operation. The FB also receives the coefficient bits so it can compute the final exponent.

The FB option also does a calculation based on the state of the initial operand as to the sign of the final results. If the result sign bit can be determined, a valid signal is sent and the sign bit is sent to the JA option. This information can be used if the JA is processing a jump on a sign bit instruction. This calculation can be done only for a scalar floating-point add instruction.

The FB option does the initial calculation to determine which exponent is larger. To detect the number of right shifts, the exponent is divided into bits 0 through 5 and 6 through 14. This way, the FA can start shifting using bits 0 through 5, and the full shift count can be sent from the FB option. This is done by comparing the following five conditions:

- exponent  $j =$  exponent  $k$
- exponent  $k >$  exponent  $j$
- exponent  $j >$  exponent  $k$
- exponent  $j + 1 =$  exponent  $k$
- exponent  $k + 1 =$  exponent  $j$

## Determining Exponent Size

---

If the upper bits are equal, the lower 6 bits determine the shift count of the coefficient.

- $j = k (14 - 6)$  and  $j > k (0 - 5)$  then right shift  $k$  by  $j - k (0 - 5)$ 
  - $j$  040012  
 $k$  040001 Right shift coefficient  $k$  by  $12 - 1 = 11$   
 Increase  $k$  exponent by 11
- $j = k (14 - 6)$  and  $k > j (0 - 5)$  then right shift  $j$  by  $k - j (0 - 5)$ 
  - $j$  040001  
 $k$  040012 Right shift  $j$  coefficient by  $12 - 1 = 11$   
 Increase  $k$  exponent by 11

If the upper bits (6 through 14) differ by 1, the lower bits can still be used to determine the full shift count.

- $j = k+1$  ( $14 - 6$ ); that is  $j > k$  ( $14 - 6$ ) by 1 and  $j < k$  ( $0 - 5$ ) then right shift  $k$  by  $j - k$  ( $0 - 5$ )
  - $j$  040100  
 $k$  040077      Right shift  $k$  coefficient by 1  
                          Increase  $k$  exponent by 1
- $j = k+1$  ( $14 - 6$ ); that is  $j > k$  ( $14 - 6$ ) by 1 and  $j > k$  ( $0 - 5$ ) then overshift occurs.
  - $j$  040177  
 $k$  040076      Right shift  $k$  coefficient by 101 places  
                          (overshift)
- $j+1 = k$  ( $14 - 6$ ); that is  $k > j$  ( $14 - 6$ ) by 1 and  $k < j$  ( $0 - 5$ ) then right shift  $j$  by  $k - j$  ( $0 - 5$ )
  - $j$  040077  
 $k$  040100      Right shift  $j$  coefficient by 1  
                          Increase  $j$  exponent by 1
- $j+1 = k$  ( $14 - 6$ ); that is  $k > j$  ( $14 - 6$ ) by 1 and  $k > j$  ( $0 - 5$ ) then overshift will occur
  - $j$  040000  
 $k$  040177      Right shift  $k$  coefficient by 177 places  
                          (overshift)

If the upper bits differ by more than 1, the lower bits can be ignored because the effect is to zero out the coefficient of the smaller exponent. This is why only the +1 case needs to be determined for the upper bits.

- $j$  040200  
 $k$  040077      Right shift  $k$  coefficient by 177  
                          Increase  $k$  exponent by 177

Refer to Figure 66 for a floating-point add flowchart.

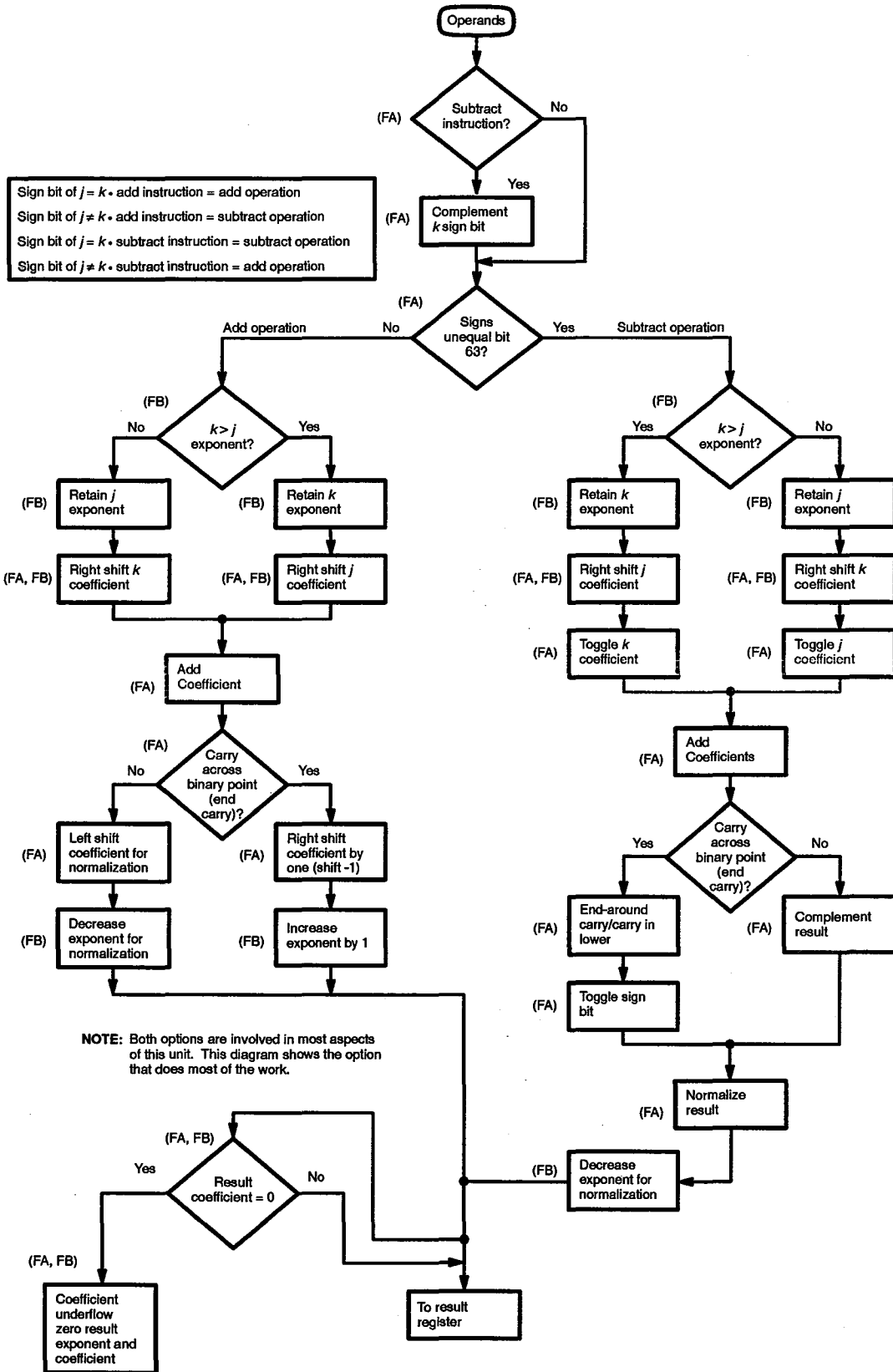
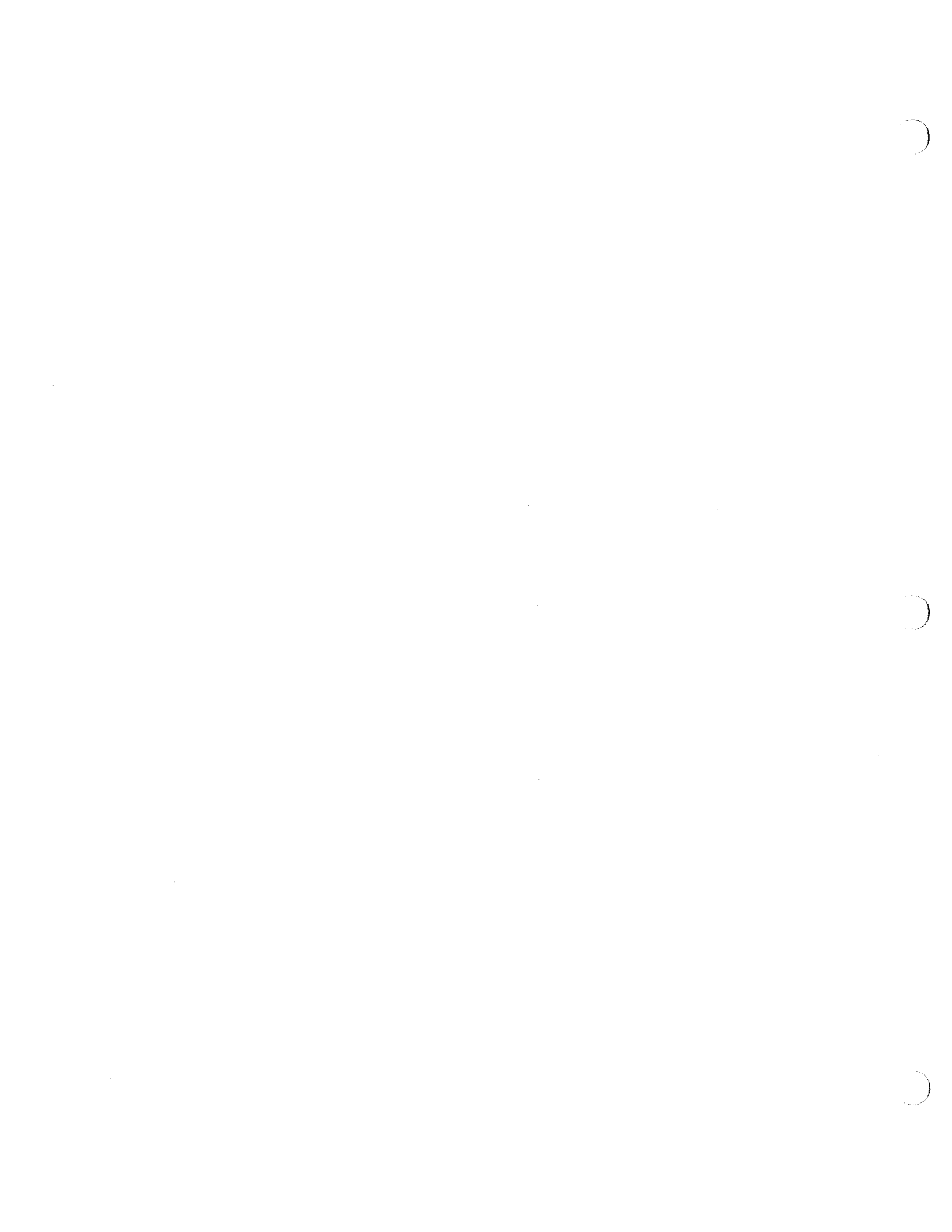


Figure 66. Floating-point Add Flowchart





# FLOATING-POINT RECIPROCAL APPROXIMATION

Refer to the following subsections for information about floating-point reciprocal approximation.

## Floating-point Division Algorithm

A CRAY T90 series computer system does not have a single functional unit dedicated to the division operation; rather, the floating-point multiply and reciprocal approximation functional units together carry out the algorithm. The following paragraphs explain the algorithm and how it is used in the functional units.

Finding the quotient of two floating-point numbers involves two steps, as shown below in the example of finding the quotient  $A/B$ .

<u>Step</u>	<u>Operation</u>
1	The B operand is sent through the reciprocal approximation functional unit to obtain its reciprocal, $1/B$ .
2	The result from Step 1 along with the A operand is sent to the floating-point multiply functional unit to obtain the product $A \times 1/B$ .

The reciprocal approximation functional unit uses an application of Newton's method for approximating the real root of an arbitrary equation,  $F(x) = 0$ , to find reciprocals.

To find the reciprocal, the equation  $F(x) = 1/x - B = 0$  must be solved. To do this, A must be found so that  $F(A) = 1/A - B = 0$ . That is, the number A is the root of the equation  $1/x - B = 0$ . The method requires an initial approximation or guess (shown as  $x_0$  in Figure 67), sufficiently close to the true root (shown as  $x_t$  in Figure 67).  $x_0$  is then used to obtain a better approximation; this is done by drawing a tangent line (line 1 in Figure 67) to the graph of  $y = F(x)$  at the point  $[x_0, F(x_0)]$ . The x-intercept of this tangent line becomes the second approximation,  $x_1$ . This process is repeated using tangent line 2 to obtain  $x_2$ , and so on.

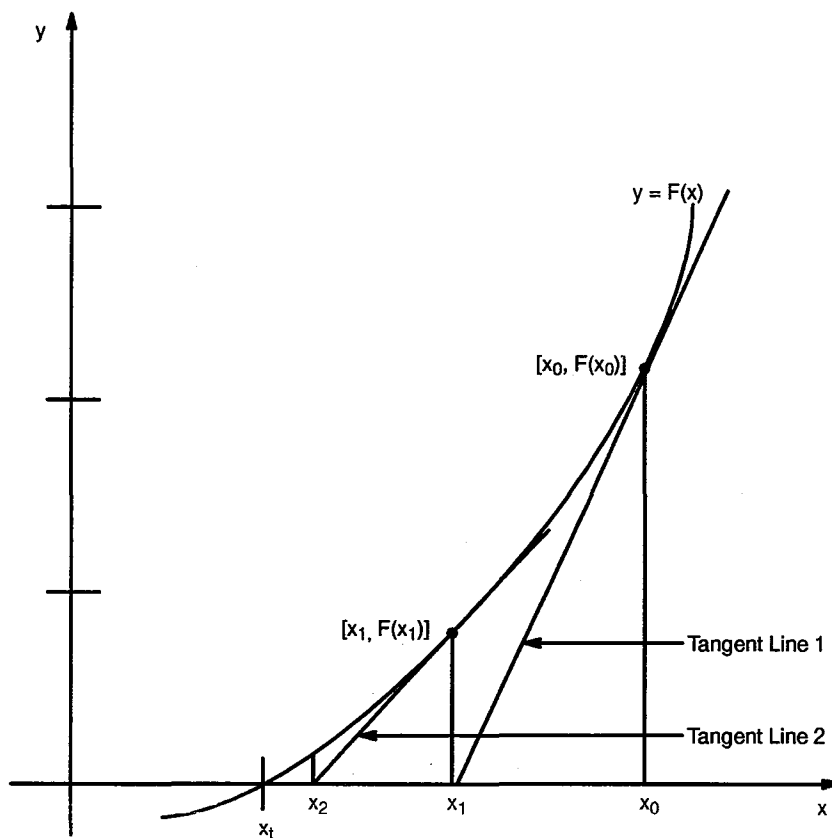


Figure 67. Newton's Method for Approximating Roots

The following iteration equation is derived from the above process:

$$x_{(i+1)} = 2x_i - x_i^2 B = x_i (2 - x_i B)$$

In the equation,  $x_{(i+1)}$  is the next iteration,  $x_i$  is the current iteration, and  $B$  is the divisor. Each  $x_{(i+1)}$  is a better approximation than  $x_i$  to the true value,  $x_t$ . The exact answer is generally not obtained at once because the correction term is not exact. The operation is repeated until the answer becomes sufficiently close for practical use.

The mainframe uses this approximation technique based on Newton's method. A hardware look-up table provides an initial guess,  $x_0$ , which is accurate to 8 bits. The following iterations are then calculated.

<u>Iteration</u>	<u>Operation</u>	<u>Description</u>
1	$x_1 = x_0(2 - x_0B)$	The first approximation is done in the reciprocal approximation functional unit and is accurate to 16 bits.
2	$x_2 = x_1(2 - x_1B)$	The second approximation is done in the reciprocal approximation functional unit and is accurate to 30 bits.
3	$x_3 = x_2(2 - x_2B)$	The third approximation is done in the floating-point multiply functional unit to calculate the correction term.

The reciprocal approximation functional unit calculates the first two iterations, while the floating-point multiply functional unit calculates the third iteration. The third iteration uses a special instruction within the floating-point multiply functional unit to calculate the correction term. This iteration is used to increase accuracy of the reciprocal approximation functional unit's answer to full precision (the floating-point multiply functional unit can provide both full- and half-precision results).

The reciprocal iteration is designed for use once with each half-precision reciprocal generated. If the third iteration (the iteration performed by the floating-point multiply functional unit) results in an exact reciprocal, or if an exact reciprocal is generated by some other method, performing another iteration results in an incorrect final reciprocal. A fourth iteration should not be done.

An example of calculating the reciprocal of 2 is provided below. Values from the look-up table in Table 24 are used.

$$\begin{aligned}
 B &= 2, \text{ start with} \\
 A_0 &= 0.2 \\
 A_1 &= 2(0.2) - (0.2)^2_2 \\
 &= 2(0.491602) - (0.491602)^2_2 \\
 &= 0.4 - 0.08 \\
 &= 0.983204 - 0.483345 \\
 &= 0.32 \\
 &= 0.499859 \\
 A_2 &= 2(0.32) - (0.32)^2_2 \\
 &= 2(0.499859) - (0.499859)^2_2 \\
 &= 0.64 - 0.2048 \\
 &= 0.999718 - 0.499718 \\
 &= 0.4352 \\
 &= 0.50000 \\
 A_3 &= 2(0.4352) - (0.4352)^2_2 \\
 &= 2(0.5) - (0.5)^2_2 \\
 &= 0.8704 - 0.378798 \\
 &= 1.0 - 0.5 \\
 &= 0.491602 \\
 &= 0.5
 \end{aligned}$$

Table 24. Reciprocal Approximation Values

B	$A_0$	$A_0^2$	$-2A_0$
1.000	0.776	0.774004	0.000
1.004	0.772	0.764044	0.010
1.010	0.766	0.754144	0.020
1.014	0.762	0.744304	0.030
1.020	0.756	0.734504	0.040
1.024	0.752	0.724744	0.050
1.030	0.750	0.721100	0.054
1.034	0.744	0.711420	0.064
1.040	0.740	0.702000	0.074
1.044	0.734	0.672420	0.104
1.050	0.732	0.666644	0.110
1.054	0.726	0.657344	0.120
1.060	0.722	0.650104	0.130
1.064	0.720	0.644400	0.134
1.070	0.714	0.635220	0.144
1.074	0.710	0.626100	0.154
1.100	0.706	0.622444	0.160
1.104	0.702	0.613404	0.170
1.110	0.700	0.610000	0.174
1.114	0.674	0.601020	0.204
1.120	0.672	0.575444	0.210
1.124	0.666	0.566544	0.220
1.130	0.664	0.563220	0.224
1.134	0.660	0.554400	0.234
1.140	0.656	0.551104	0.240
1.144	0.652	0.542344	0.250
1.150	0.650	0.537100	0.254
1.154	0.646	0.533644	0.260
1.160	0.642	0.525204	0.270
1.164	0.640	0.522000	0.274
1.170	0.636	0.516604	0.300
1.174	0.632	0.510244	0.310
1.200	0.630	0.505100	0.314
1.204	0.626	0.501744	0.320
1.210	0.624	0.476620	0.324
1.214	0.620	0.470400	0.334
1.220	0.616	0.465304	0.340
1.224	0.614	0.462220	0.344
1.230	0.612	0.457144	0.350
1.234	0.610	0.454100	0.354
1.240	0.604	0.446020	0.364
1.244	0.602	0.443004	0.370
1.250	0.600	0.440000	0.374

Table 24. Reciprocal Approximation Values (continued)

B	$A_0$	$A_0^2$	$-2A_0$
1.254	0.576	0.435004	0.400
1.260	0.574	0.432020	0.404
1.264	0.572	0.427044	0.410
1.270	0.570	0.424100	0.414
1.274	0.566	0.421144	0.420
1.300	0.564	0.416220	0.424
1.304	0.562	0.413304	0.430
1.310	0.560	0.410400	0.434
1.314	0.556	0.405504	0.440
1.320	0.554	0.402620	0.444
1.324	0.552	0.377744	0.450
1.330	0.550	0.375100	0.454
1.334	0.546	0.372244	0.460
1.340	0.544	0.367420	0.464
1.344	0.542	0.364604	0.470
1.350	0.540	0.362000	0.474
1.354	0.536	0.357204	0.500
1.360	0.534	0.354420	0.504
1.364	0.532	0.351644	0.510
1.370	0.530	0.347100	0.514
1.374	0.526	0.344344	0.520
1.400	0.524	0.341620	0.524
1.404	0.522	0.337104	0.530
1.410	0.520	0.334400	0.534
1.414	0.520	0.334400	0.534
1.420	0.516	0.331704	0.540
1.424	0.514	0.327220	0.544
1.430	0.512	0.324544	0.550
1.434	0.510	0.322100	0.554
1.440	0.506	0.317444	0.560
1.444	0.506	0.317444	0.560
1.450	0.504	0.315020	0.564
1.454	0.502	0.312404	0.570
1.460	0.500	0.310000	0.574
1.464	0.476	0.305404	0.600
1.470	0.476	0.305404	0.600
1.474	0.474	0.303020	0.604
1.500	0.472	0.300444	0.610
1.504	0.470	0.276100	0.614
1.510	0.470	0.276100	0.614
1.514	0.466	0.273544	0.620
1.520	0.464	0.271220	0.624
1.524	0.462	0.266704	0.630

Table 24. Reciprocal Approximation Values (continued)

B	$A_0$	$A_0^2$	$-2A_0$
1.530	0.462	0.266704	0.630
1.534	0.460	0.264400	0.634
1.540	0.456	0.262104	0.640
1.544	0.456	0.262104	0.640
1.550	0.454	0.257620	0.644
1.554	0.452	0.255344	0.650
1.560	0.452	0.255344	0.650
1.564	0.450	0.253100	0.654
1.570	0.446	0.250644	0.660
1.574	0.446	0.250644	0.660
1.600	0.444	0.246420	0.664
1.604	0.442	0.244204	0.670
1.610	0.442	0.244204	0.670
1.614	0.440	0.242000	0.674
1.620	0.436	0.237604	0.700
1.624	0.436	0.237604	0.700
1.630	0.434	0.235420	0.704
1.634	0.434	0.235420	0.704
1.640	0.432	0.233244	0.710
1.644	0.430	0.231100	0.714
1.650	0.430	0.231100	0.714
1.654	0.426	0.226744	0.720
1.660	0.426	0.226744	0.720
1.664	0.424	0.224620	0.724
1.670	0.422	0.222504	0.730
1.674	0.422	0.222504	0.730
1.700	0.420	0.220400	0.734
1.704	0.420	0.220400	0.734
1.710	0.416	0.216304	0.740
1.714	0.416	0.216304	0.740
1.720	0.414	0.214220	0.744
1.724	0.412	0.212144	0.750
1.730	0.412	0.212144	0.750
1.734	0.410	0.210100	0.754
1.740	0.410	0.210100	0.754
1.744	0.406	0.206044	0.760
1.750	0.406	0.206044	0.760
1.754	0.404	0.204020	0.764
1.760	0.404	0.204020	0.764
1.764	0.402	0.202004	0.770
1.770	0.402	0.202004	0.770
1.774	0.400	0.200000	0.774



## Handling of B Exponent

---

The following example show how the floating-point reciprocal approximation unit handles the B exponent:

B =	40000 + E	1XXXXX	XXXXXX	XXXXXX
	Exponent	Coefficient		

Value of  $B = 2^E \times 0.1XXX$  — X Normalize floating-point number

$$B = 2^{E-1} \times 1.XXX \text{ — X} \quad \text{Left shift by 1}$$

$$\text{Let } b = 1.XXX \text{ — X}$$

$$\text{then } B = 2^{E-1} \times b$$

$$\frac{1}{B} = \frac{1}{2^{E-1} \times b} = \frac{1}{2^{E-1}} \times \frac{1}{b}$$

$$\text{Let } n = E - 1$$

$$\frac{1}{2^n} = \frac{2^{-n}}{1} \text{ OR } \frac{1}{2^{E-1}} = \frac{2^{-(E-1)}}{1} = \frac{2^{-E+1}}{1}$$

$$\frac{1}{B} = \frac{2^{-E+1}}{1} \times \frac{1}{b}$$

The following method is used in the CRAY T90 series system:

51132 Exponent

Perform 1's complement 26645

1	Add one for normalization
1	Add one for two's complement
<u>26647</u>	

## Floating-point Reciprocal Approximation Instructions

---

Refer to Table 25 for a list of the floating-point reciprocal approximation instructions. Figure 68 is an illustration of the reciprocal approximation functional unit.

Table 25. Floating-point Reciprocal Approximation Instructions

Instruction	CAL	Description
070j0	$S_i/HS_j$	Floating-point reciprocal approximation of $(S_j)$ to $S_i$
174j0	$V_i/HV_j$	Floating-point reciprocal approximation $(V_j)$ to $V_i$
030i0k	$A_iA_k$	Transmit $A_k$ to $A_i$

### RA Option

There is one RA option used; it is the first option in the reciprocal approximation functional unit. It performs all of the vector pop operations as well as the exponent, floating-point range error, look-up table and first iteration of the reciprocal function. The RA receives and decodes the control necessary to gate the data to the correct unit and generates the control for the rest of the reciprocal approximation functional unit.

### RB Option

There is one RB option used; it is the second option in the reciprocal approximation functional unit. The RB option gets the  $A_1$  iteration data from the RA option and performs the  $A_1^2$  function to send it to the RC option final iteration pyramid. The B2 operand data is also delayed on the RB option before being sent to the RC.

When the  $A_1^2$  and the B2 data is available, the RB option generates the jagged portion of the A2 pyramid. After a couple of levels of adds, those bits are sent to the RC option to be included in the rest of the pyramid.

### RC Option

The RC option is the last option in the unit. It performs the final iteration of the reciprocal approximation function. It receives the  $A_1^2$ ,  $A_1$ , and B2 data from the RB option; forms the pyramid; and adds all the data to get A2. The outputs of the RC option are all forced to 0's by the input control during any operation of the vector pop unit.

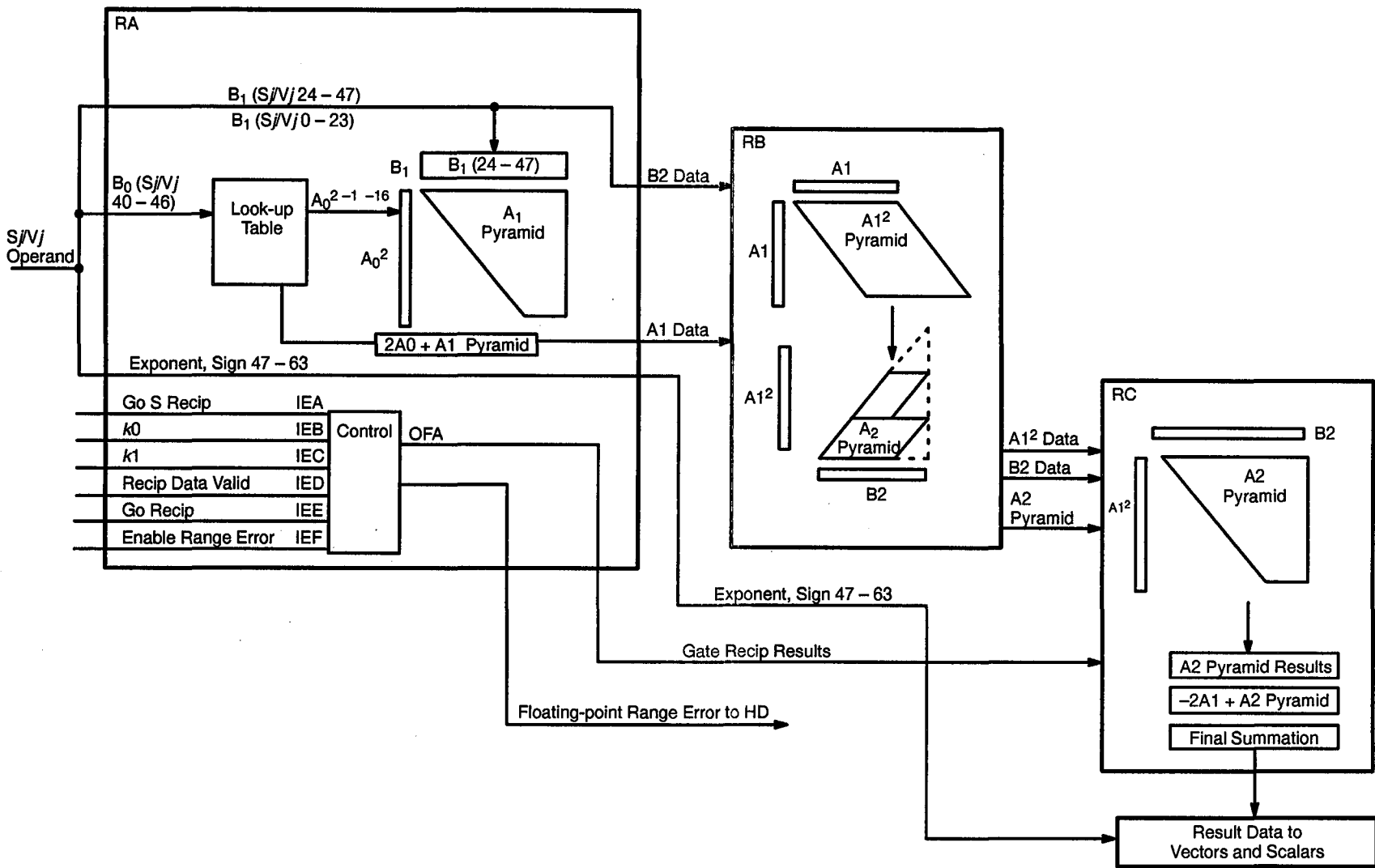


Figure 68. Reciprocal Approximation Functional Unit

## Multiply Algorithm

The reciprocal approximation functional unit uses a recode multiply algorithm known as Booth Recode algorithm. It is used on several pieces of the various pyramids. This algorithm was used instead of the standard pyramid formations to save space on the options and make them easier to route.

C

C

C

## FLOATING-POINT MULTIPLY

The scalar and vector registers share the floating-point multiply functional unit. Two floating-point operands are sent to the multiply functional unit by either the scalar or the vector registers. The signs of the two operands are combined through an exclusive OR operation, the exponents are added together, and the two 48-bit coefficients are multiplied. Multiplying two 48-bit numbers produces a 96-bit result. Because the result register (either a scalar or a vector register) can hold only 48 bits in the coefficient, only the upper 48 bits of the 96-bit result are kept. The lower 48 bits are lost; in fact, most are not generated.

The floating-point multiply functional unit also passes operands to the AM option for the integer multiply operation.  $S_j$  and  $V_k$  data are relayed through the NA and NB options for use by the AM option during integer multiply operations. The floating-point multiply functional unit no longer performs integer multiply.

The floating-point multiply functional unit can also be used to generate a third iteration in conjunction with the reciprocal approximation functional unit. Generating the third iteration creates a full-precision coefficient, utilizing all 48 bits of the coefficient. The full-precision reciprocal number can then be multiplied by the multiplier to finish the division. If full precision is not needed, then there is no need to generate a third iteration. Instead, the results from the reciprocal approximation functional unit are multiplied by the multiplier using a multiply instruction. The following multiply instructions add 2 rounding bits and truncate the lower 19 bits of the coefficient:  $065ijk$ ,  $162ijk$ , or  $163ijk$ .

The floating-point multiply functional unit has the same range error conditions as the floating-point add. If an overflow condition exists, the floating-point number has exceeded the limits of the computer system. When an overflow condition occurs, the result register receives the calculated coefficient with an exponent forced to  $60000_8$ . An overflow condition also causes a flag to be set in the exchange package if the interrupt on floating-point error mode bit is set. An underflow condition exists when the result exponent is equal to or less than  $17777_8$ . When an underflow condition exists, both the final exponent and the coefficient are forced to 0's, but no flag sets in the exchange package.

The floating-point multiply functional unit performs the 064*ijk* through 067*ijk* instructions for the scalar registers and performs the 160*ijk* through 167*ijk* instructions for the vector registers. Because the multiply unit is shared by both the scalar and vector registers, a functional unit reservation must be checked before one of these instructions can issue.

The floating-point multiply unit is controlled by the mode bits, which are taken from *h* field bits 1 and 0 for the 064*ijk* through 067*ijk* instructions, or from *h* field bits 2 and 1 for the 160*ijk* through 167*ijk* instructions. The 064*ijk* instruction, which is the scalar equivalent of the 160*ijk* and 161*ijk* instructions for the vector registers, performs a floating-point multiply of two scalar registers.

The 065*ijk* instruction, which is the equivalent of the 162*ijk* or 163*ijk* instruction for vector registers, is used with the reciprocal approximation functional unit to complete a divide sequence. In other words, a 065*ijk* instruction would be issued after a 070*ijk* instruction. The 065*ijk* instruction adds 2 bits into the final summation in bit positions 16 and 17. These 2 bits are called *strong rounding bits* because they have a major effect on the answer. When the final summation is completed, the 065*ijk* instruction also causes the lower 19 bits to be truncated; the control term that enables this is called *strong round*.

The 066*ijk* instruction, which is the equivalent of the 164*ijk* through 165*ijk* instruction for the vector register, is used only after the third iteration has been completed within the floating-point multiply functional unit. The 066*ijk* instruction generates 2 *weak rounding bits*. These 2 bits are called *weak rounding bits* because they are added into the lower portion of the summation, having only a minimal effect on the final summation.

The 067*ijk* instruction, which is the equivalent of the 167*ijk* instruction for the vector registers, forms part of the third iteration as follows.

The third iteration is equal to  $A_3 = (2A_2 - A_2^2B)$ . The 067*ijk* instruction solves for  $(-2 + A_2 * B)$  by first multiplying  $A_2$  times  $B$ , and then adding  $-2$  to the product. The  $-2$  addition is accomplished by adding 1 to each sum in bit position 0 through 46 during the summation of  $(A_2 * B)$ . These 1 bits actually comprise 49 1 bits and are generated by the control terms, which are decoded from a 067*ijk* or a 167*ijk* instruction.

The 067*ijk* instructions also complement or toggle their final result to convert  $-A_3 = (-2 + A_2 * B)$  to  $A_3 = (2 - A_2 * B)$ . At this point, the 064*ijk* instruction completes the third iteration by multiplying  $A_2$  times the result of the 067*ijk* instruction. In other words,  $A_2 * (2 - A_2 * B) = (2A_2 - A_2^2B)$ . In conclusion, the 067*ijk* instruction,

along with the 064ijk instruction, generates the third iteration equation  $A_3 = (2A_2 - A_2^2B)$ .

## Divide Sequence

A divide sequence produces an answer accurate to 29 places. The instructions used to perform this divide sequence are shown below. If an answer accurate to 48 places is required, a software algorithm (shown below) produces the desired results.

$$\underline{S6 = S1/S2}$$

Accurate to 29 Bits:

#1	070320	$S3 = 1/S2$
#2	065613	$S6 = S1 * FS3$

Accurate to 48 Bits:

$$\underline{S6 = S1/S2}$$

#1	070320	$S3 = 1/S2$
#2	067432	$S4 = (2 - [S3*S2])$
#3	064543	$S5 = S4*S3$
#4	066651	$S6 = S5*S1$
#1	$A_1 = 2A_0 - A_0^2B$	First Iteration
	$A_2 = 2A_1 - A_1^2B$	Second Iteration



#2       $S4 = (2 - (A_2 * B))$       Third Iteration

#3       $A3 = A_2(2 - (A_2 * B))$

or

$A3 = 2A_2 - A_2^2B$

#4       $S6 = A_3 * S1$       Third Iteration \* S1

## Floating-point Multiply Functional Unit Instructions

Refer to Table 26 for a list of the floating-point multiply functional unit instructions.

Table 26. Floating-point Multiply Functional Unit Instructions

Instruction	CAL	Description
064ijk	$SS_j^*FSk$	Scalar floating-point product of (Sj) times (Sk) to (Si)
065ijk	$SS_j^*HSk$	Scalar floating-point product, half precision, (Sj) times (Sk) to (Si)
066ijk	$SS_j^*RSk$	Scalar floating-point product, full precision, (Sj) times (Sk) to (Si)
067ijk	$SS_j^*ISk$	Scalar floating-point product, 2 minus the product of (Sj) times (Sk) to (Si)
160ijk	$VS_j^*FVk$	Vector floating-point product (Sj) times (Vk elements) to Vi
161ijk	$VN_j^*FVk$	Vector floating-point product (Vj elements) times (Vk elements) to Vi
162ijk	$VS_j^*HVK$	Half precision, (Sj) times (Vk elements) to Vi
163ijk	$VN_j^*HVK$	Half precision, (Vj elements) times (Vk elements) to Vi
164ijk	$VS_j^*RVk$	Full precision, (Sj) times (Vk elements) to Vi
165ijk	$VN_j^*RVk$	Full precision, (Vj elements) times (Vk elements) to Vi
166ijk	$VS_j^*Vk$	32-bit integer products of (Sj) and (Vk) to Vi (C90 mode)
167ijk	$VN_j^*Vk$	Iteration, two minus (Vj elements) times (Vk elements) to Vi

Because this is a dual-pipe functional unit, there are two options. The even elements are processed by pipe 0, which is option number 000; and the odd elements are processed by pipe 1, which is option number 001.

### NA Option

The NA option forms the upper right portion of the pyramid. The pyramid is 24 bits deep from sum bits 40 to 65. It is generated from  $j$  operand bits 17 through 47, and  $k$  operand bits 0 through 41. The scalar  $j/k$  and vector  $j/k$  operands are multiplexed (muxed) before the pyramid is formed.

The NA option relays a copy of  $S_j$  bits 40 through 47 and  $V_k$  bits 0 through 41 to the AM option for the 166 instruction (integer multiply).

### NB Option

The NB option forms the lower right portion of the pyramid. The pyramid increments from 17 bits deep at sum bit 40, to 24 bits deep at sum bit 47, and then tapers down to 6 bits deep at sum bit 65. It remains at 9 bits from sum bit 65 to sum bit 78.

It is generated from  $j$  operand bits 0 through 39 and  $k$  operand bits 24 through 47. The scalar  $j/k$  and vector  $j/k$  operands are muxed before the pyramid is formed.

The NB option also forms rounding bits for all floating-point multiply instructions at sum bits 78 through 40. The first two-level results are then sent to the ND option for final summation.

The NB option relays a copy of  $S_j$  bits 0 through 39 and  $V_k$  bits 42 through 47 to the AM option for the 166 instruction (integer multiply). The NB option also sends the control signal  $Go V 166$  to the AM option.

### NC Option

The NC option forms the lower left portion of the pyramid. The pyramid decrements from 20 bits deep at sum bit 66, to 8 bits deep at sum bit 78. The pyramid then starts from 16 bits deep at sum bit 79 and tapers to 1 bit deep at sum bit 94.

The pyramid is generated from  $j$  operand bits 28 through 62 and  $k$  operand bits 16 through 47. The scalar  $j/k$  and vector  $j/k$  operands are muxed before the pyramid is formed. The NC option also forms rounding bits for all floating-point multiply instructions at sum bits 79 through 94. The first two-level results are then sent to the ND option for final summation.

The NC option also computes the exponent, underflow, and range error. The exponent value is sent to the ND option to compute the exponent  $-1$  and to multiplex the correct exponent. The NC option also computes the final sign bit and sends it to the result register. The NC sends the sign bit back to the JA for possible early branch determination.

The NC option relays a copy of  $S_j$  bits 48 through 62 to the AM option for the 166 instruction (integer multiply).

## ND Option

The ND option does the final summation for the floating-point multiply pyramid. The ND sends the final coefficient and exponent to the result registers. The NC also transmits the range error signal to the HD option.

Refer to Figure 69 for a block diagram of floating-point multiply and to Figure 70 for an illustration of the floating-point multiply first-level summation.

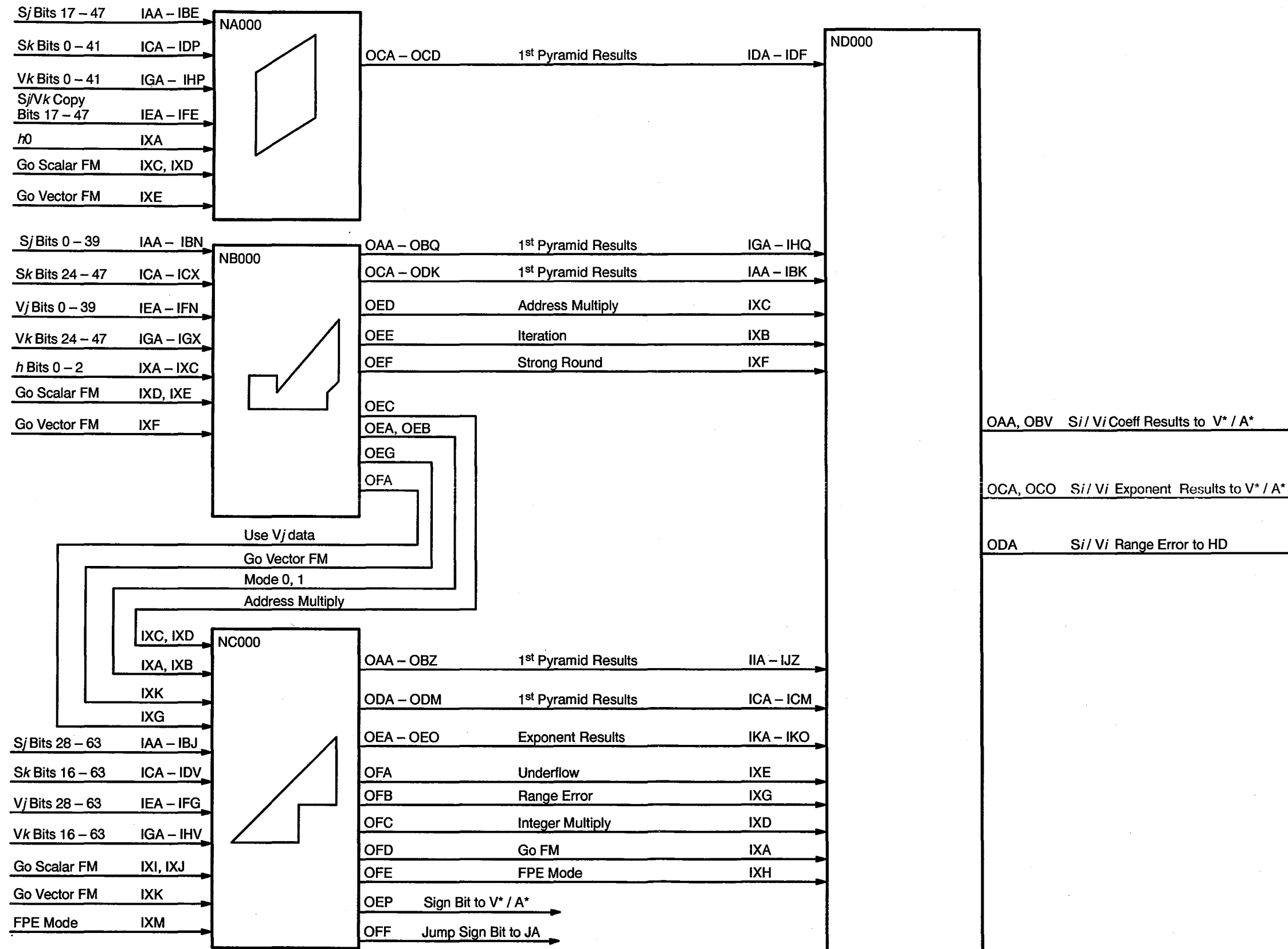


Figure 69. Floating-point Multiply Block Diagram

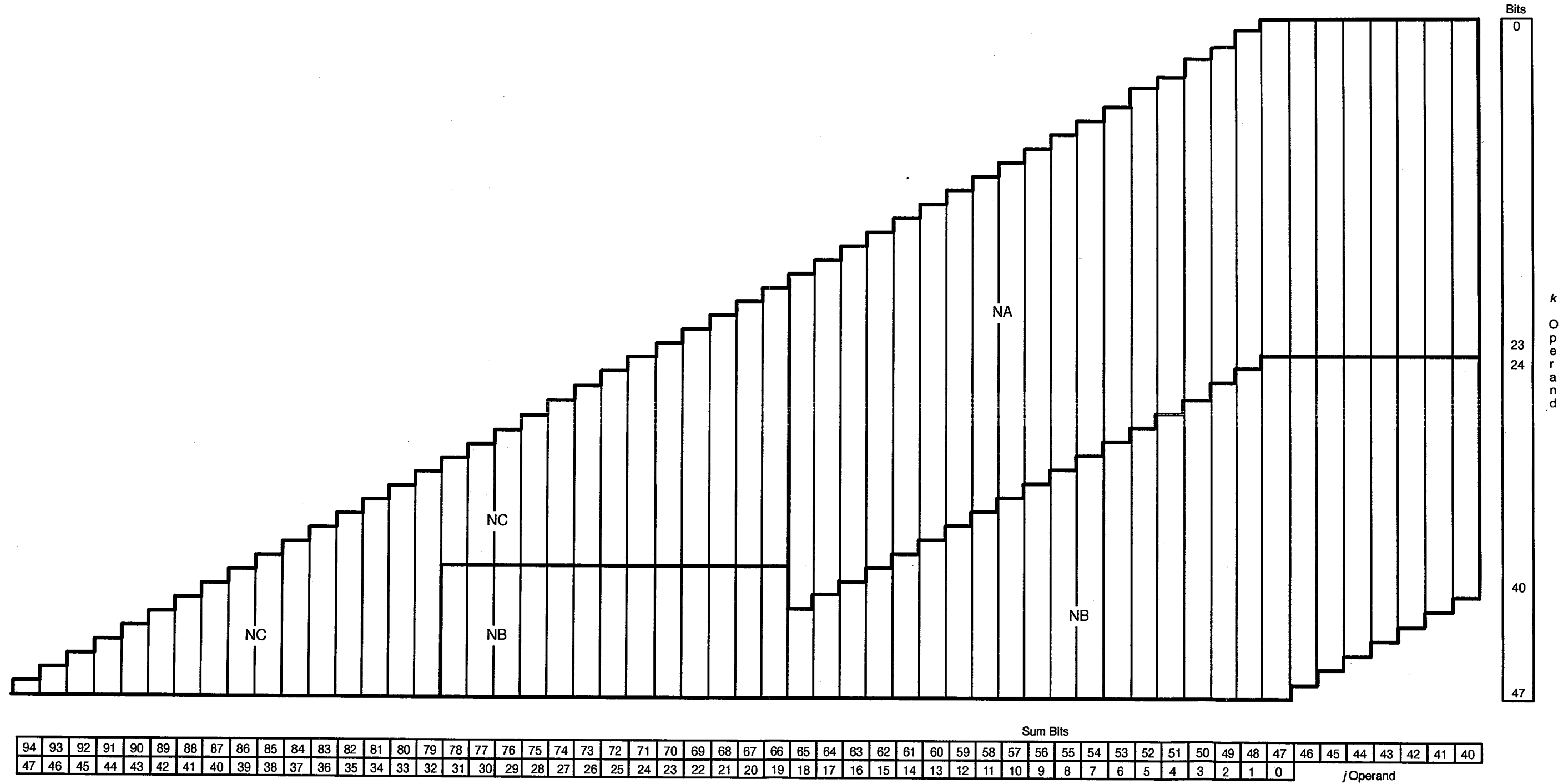


Figure 70. Floating-point Multiply First-level Summation

# BIT MATRIX MULTIPLY

The OA option performs the bit matrix multiply operation. The functional unit consists of six OA options.

The OA option performs two functions related to bit matrix multiply. The first function is to load the B array with the  $V_j$  operand. The second function is to perform the  $A \times B^T$  operation where A is either the  $S_j$  or  $V_j$  operand and  $B^T$  is the B array transposed. The scalar operation produces a scalar result, and the vector operation produces a vector result.

Each OA option receives 22 bits of the operand. OA002 and OA005 receive 20 bits, and the last two inputs are forced to zero. Each OA option holds 32 elements x 22 bits. When performing the  $A \times B^T$  operation, each OA produces a partial result for each of the 32 elements. The partial results are then sent the appropriate OA option to complete the final results. There is only one copy of each control bit coming into the functional unit, so OA001 and OA004 relay the control bits to the other options.

## Bit Matrix Multiply Theory of Operation

The bit matrix multiply (BMM) functional unit performs a logical multiplication of two matrices, designated A and B, resulting in a single-bit result for each pair of elements multiplied. The matrices, which are held in vector registers, may vary in size from 1 bit x 1 bit (1 x 1) to 64 x 64 bits. The size of the matrix is specified by the vector length (VL) register (example: VL = 20 specifies 20 x 20 matrices).

The following conditions are necessary to obtain valid results:

- The two matrices must be square and of equal size.
- The two matrices must be left-justified in the vector registers to element 0, bit 63.
- Unused bits of each element that contain part of the matrix must be zeroed.
- Elements not containing parts of a matrix are unaffected.

Result matrix C is the product of matrix A and matrix B transposed ( $B^t$ ).  $B^t$  is formed from matrix B by interchanging its rows and columns.

In addition to performing full 64 x 64 matrix multiply operations, the BMM functional unit performs a scalar-vector multiply operation and stores the result in an S register.

Figure 71 is an illustration of 20 x 20 and 50 x 50 matrices as stored in vector registers.

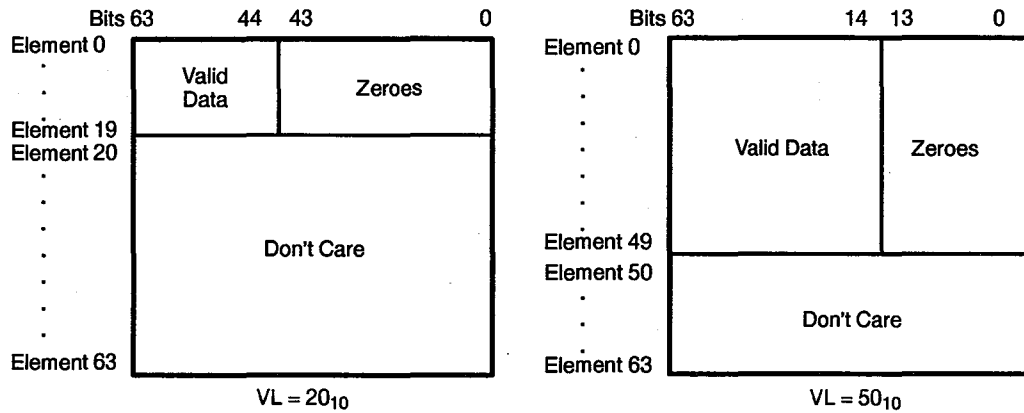


Figure 71. Vector Storage of Bit Matrices

In this section, the notation used to represent individual bits of a matrix consists of a lower-case letter followed by a subscripted numeric field. The letter represents the name of the matrix; the numerics denote, respectively, the element and bit of the vector register data. Elements and bits numbered from 1 to 9 are represented as a 2-digit number; elements and bits numbered upward from 10 are separated by a comma. For example:

$a_{3,7}$  represents matrix A, element 3, bit 7

$b_{15,43}$  represents matrix B, element 15, bit 43

$a_{3,12}$  represents matrix A, element 3, bit 12

Mathematically, matrices A and B can then be represented as shown in Figure 72. Note that the ultimate degree of both element and bit can be represented by  $n$  because these must be square matrices. Each row of a matrix corresponds to an element of a vector register.

$$A = \begin{vmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \cdot & \cdot & \cdot & & \cdot \\ \cdot & \cdot & \cdot & & \cdot \\ \cdot & \cdot & \cdot & & \cdot \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{vmatrix} \quad B = \begin{vmatrix} b_{11} & b_{12} & b_{13} & \dots & b_{1n} \\ b_{21} & b_{22} & b_{23} & \dots & b_{2n} \\ \cdot & \cdot & \cdot & & \cdot \\ \cdot & \cdot & \cdot & & \cdot \\ \cdot & \cdot & \cdot & & \cdot \\ b_{n1} & b_{n2} & b_{n3} & \dots & b_{nn} \end{vmatrix}$$

Figure 72. Mathematical Representation of Matrices A and B

The BMM functional unit transposes matrix B as it is loaded into the BMM storage area. The elements (rows) of the B matrix data are interchanged with the bit positions (columns) as shown in Figure 73.

$$B = \begin{vmatrix} b_{11} & b_{12} & b_{13} & \dots & b_{1n} \\ b_{21} & b_{22} & b_{23} & \dots & b_{2n} \\ b_{31} & b_{32} & b_{33} & \dots & b_{3n} \\ \cdot & \cdot & \cdot & & \cdot \\ \cdot & \cdot & \cdot & & \cdot \\ b_{n1} & b_{n2} & b_{n3} & \dots & b_{nn} \end{vmatrix} \quad B^t = \begin{vmatrix} b_{11} & b_{21} & b_{31} & \dots & b_{n1} \\ b_{12} & b_{22} & b_{32} & \dots & b_{n2} \\ b_{13} & b_{23} & b_{33} & \dots & b_{n3} \\ \cdot & \cdot & \cdot & & \cdot \\ \cdot & \cdot & \cdot & & \cdot \\ b_{1n} & b_{2n} & b_{3n} & \dots & b_{nn} \end{vmatrix}$$

Figure 73. B Matrix and B<sup>t</sup> Matrix Relationships



The product  $C = AB^t$  is defined as shown in Figure 74.

$$AB^t = \begin{array}{c|cccc|cccc|cccc|}
 & a_{11} & a_{12} & a_{13} & \dots & a_{1n} & b_{11} & b_{21} & b_{31} & \dots & b_{n1} & c_{11} & c_{12} & c_{13} & \dots & c_{1n} \\
 & a_{21} & a_{22} & a_{23} & \dots & a_{2n} & b_{12} & b_{22} & b_{32} & \dots & b_{n2} & c_{21} & c_{22} & c_{23} & \dots & c_{2n} \\
 & a_{31} & a_{32} & a_{33} & \dots & a_{3n} & b_{13} & b_{23} & b_{33} & \dots & b_{n3} & c_{31} & c_{32} & c_{33} & \dots & c_{3n} \\
 & \cdot & \cdot & \cdot & & \cdot & \cdot & \cdot & \cdot & & \cdot & \cdot & \cdot & \cdot & & \cdot \\
 & \cdot & \cdot & \cdot & & \cdot & \cdot & \cdot & \cdot & & \cdot & \cdot & \cdot & \cdot & & \cdot \\
 & \cdot & \cdot & \cdot & & \cdot & \cdot & \cdot & \cdot & & \cdot & \cdot & \cdot & \cdot & & \cdot \\
 & a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} & b_{1n} & b_{2n} & b_{3n} & \dots & b_{nn} & c_{n1} & c_{n2} & c_{n3} & \dots & c_{nn} \\
 & A & & & & & B^t & & & & & C & & & & 
 \end{array}$$

where:

$$\begin{aligned}
 C_{11} &= a_{11}b_{11} \oplus a_{12}b_{12} \oplus a_{13}b_{13} \oplus \dots \oplus a_{1n}b_{1n} \dagger \\
 C_{12} &= a_{11}b_{21} \oplus a_{12}b_{22} \oplus a_{13}b_{23} \oplus \dots \oplus a_{1n}b_{2n} \\
 C_{13} &= a_{11}b_{31} \oplus a_{12}b_{32} \oplus a_{13}b_{33} \oplus \dots \oplus a_{1n}b_{3n} \\
 &\cdot \\
 &\cdot \\
 C_{21} &= a_{21}b_{11} \oplus a_{22}b_{12} \oplus a_{23}b_{13} \oplus \dots \oplus a_{2n}b_{1n} \\
 &\cdot \\
 &\cdot \\
 C_{32} &= a_{31}b_{21} \oplus a_{32}b_{22} \oplus a_{33}b_{23} \oplus \dots \oplus a_{3n}b_{2n} \\
 &\cdot \\
 &\cdot
 \end{aligned}$$

†  $\oplus$  indicates an exclusive OR operation.

Figure 74. Multiplication of A and B<sup>t</sup>

## Instructions

---

Refer to Table 27 for a list of the bit matrix multiply instructions.

Table 27. Bit Matrix Multiply Instructions

Instruction	CAL	Description
1740j4	BMM LVj	Transmit Vj elements 0 – 63 to B matrix
1740j5	BMM UVj	Transmit Vj elements 64 – 127 to B matrix
174j6	Vi Vj* BT	Transmit the value of Vj multiplied by the transposed B matrix to Vi
070j6	Si Sj* BT	Transmit the value of Sj multiplied by the transposed B matrix to Si
002210	CBL	Clear the bit matrix loaded (BML) flag

Refer to Figure 75 for a BMM block diagram for pipe 0 and to Figure 76 for a BMM block diagram for pipe 1.



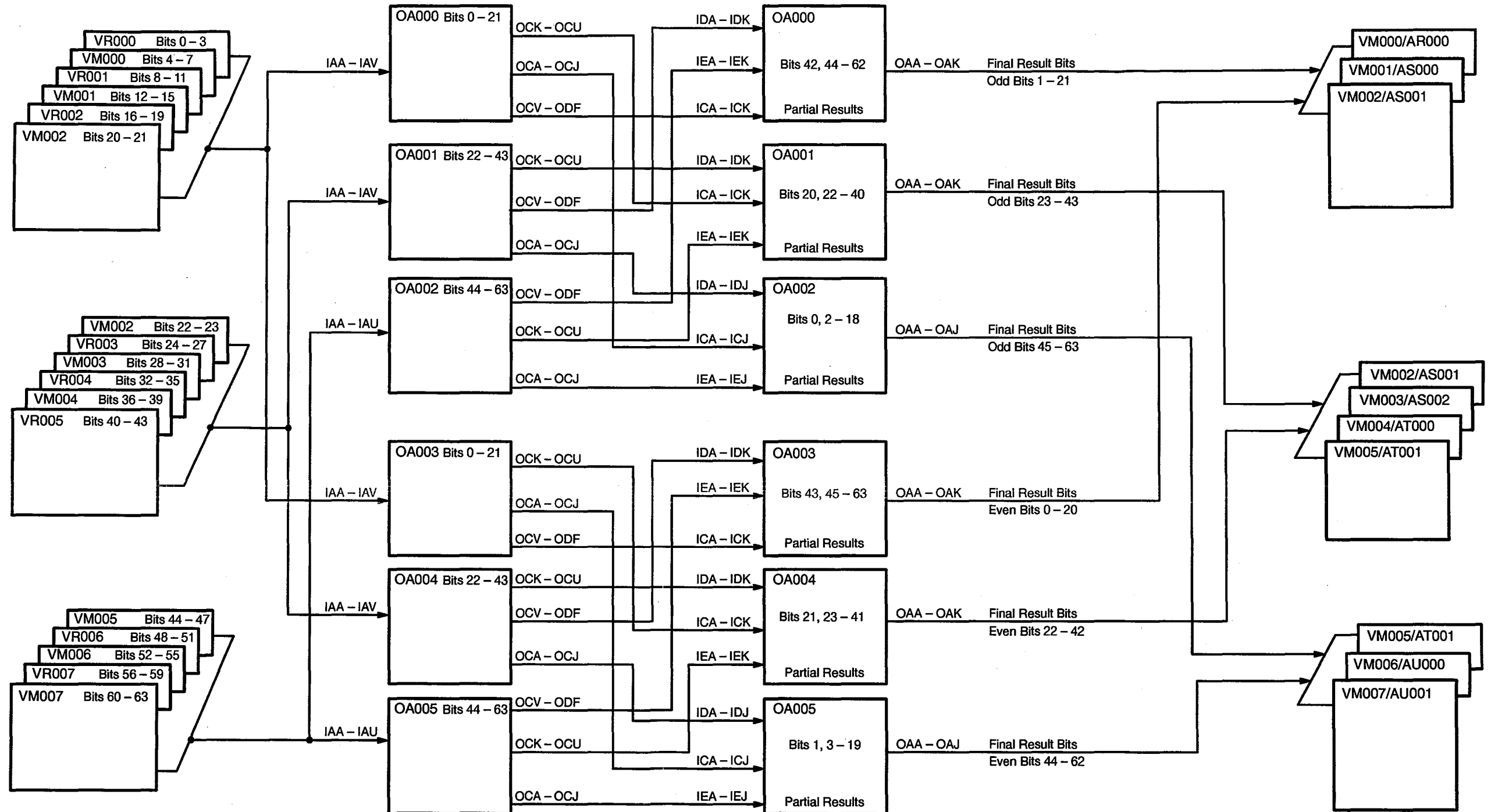


Figure 75. Bit Matrix Multiply Block Diagram Pipe 0

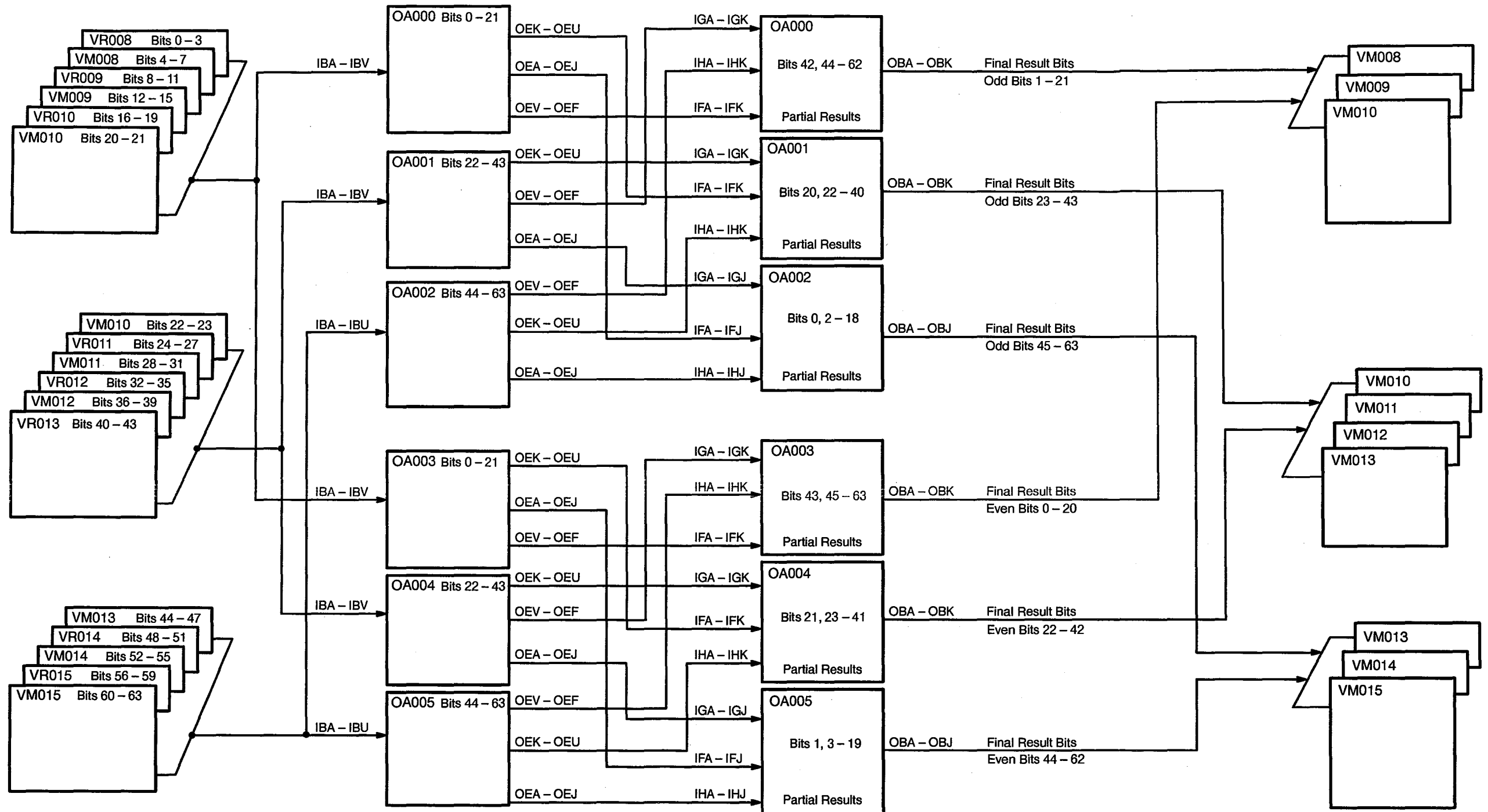


Figure 76. Bit Matrix Multiply Block Diagram Pipe 1

# INSTRUCTION BUFFERS

The instruction buffers are located on four IC options; Table 28 shows how the four IC options are partitioned. Each IC option contains 8 buffers, and each buffer holds 32 16-bit words. The IC options also hold data for functions other than instructions.

Table 28. IC Options

Bit Type	IC000	IC001	IC002	IC003
Instruction data bits	0 – 7 and 32 – 39	8 – 15 and 40 – 47	16 – 23 and 48 – 55	24 – 31 and 56 – 63
B address bits	0 – 7	8 – 15	16 – 23	24 – 31
Fetch address bits	0 – 7	8 – 15	16 – 23	24 – 31
Logical address translation (LAT) address bits	0 – 7 and 32 – 39	8 – 15 and 40 – 47	16 – 23 and 48 – 55	24 – 31 and 56 – 63
Exchange P address bits	0 – 7 and 32 – 39	8 – 15 and 40 – 47	16 – 23 and 48 – 55	24 – 31 and 56 – 63
Fetch destination code fan-out bits	0, 1	2, 3	4, 5	6, 7

## Fetch

---

The IC options generate a deadstart fetch after the first 20<sub>8</sub> words have been received; this is the number of words in the exchange package. The IC option counts the number of common memory valid codes received, and this count enables the deadstart fetch signal to be generated.

When data is fetched from memory, it is requested as a block of 32 words (4 blocks of 8 words with the first word of this block being the first word that is needed). For example, if a branch is made to address 1005, that address is requested first, followed by addresses 1006 to 1037, then 1000 to 1004.

When the common memory data arrives, the IC compares the incoming code with the expected code. This code tells the IC option where to put the data in the buffer. Data can arrive at the IC from memory in any order; it is reordered inside the buffer. The memory code enables this to happen. Along with every 16 bits of memory data, a 9-bit code is also

sent. This code specifies the buffer and the element in the buffer into which the word is to be loaded. The following illustration shows a breakdown of the code.

Valid	Buffer	Element
8	7 6 5	4 3 2 1 0

The data arrives at the IC options 2 words at a time. When the data starts arriving, the IC options look for the first 4 words. These words go through a bypass path, to the read-out registers, and then to the JA options for issue.

Two pointers are associated with bypass: a read pointer and a write pointer. As long as the write pointer stays ahead of read issue, the first 4 words will issue. The buffers will continue to fill while the first 4 words are issuing. If the first 4 words issue and the buffers are not full, then issue stops until the buffers fill and the buffer valid bit is set. The instruction parcels will then start leaving the buffers for the JA options.

## Prefetch

---

A prefetch is initiated when the buffer read-out pointer reaches address 30<sub>8</sub> in the buffer or a branch occurs to addresses 30 to 37<sub>8</sub>.

The prefetch checks to determine whether the next sequential buffer is already in-stack. If it is not, a fetch is initiated to the next sequential common memory address. When the count in the buffer reaches 37<sub>8</sub>, the IC advances the buffer pointer and checks to ensure that the read data valid bit is set. If the read data valid bit is not set, the IC option enables the wait first word flag and waits for the first word to be received from common memory.

**NOTE:** The prefetch will always occur, but it can be blocked or aborted by any branch sequence in progress.

Prefetch can, in some cases, cause a decrease in performance. For example, if the first word of the next sequential instruction block is needed while the current instruction block is being fetched, a delay occurs. In this case, issue stops until the last word of the next block is fetched.

If an out-of-stack branch occurs while the next sequential block is waiting to be prefetched, the prefetch is aborted and the block containing the branch address is fetched instead. Issue of instructions at the branch address are delayed until the fetch of the current block is completed and a fetch of the current block containing the branch address begins.

Another problem with prefetch occurs when executing an instruction at the top of logical address translation (LAT) space. The code may execute a branch to lower memory but the prefetch may try to initiate a fetch from the next sequential memory location. If the next sequential memory location is out of the LAT range, a range error may occur. This will happen if the branch is within 8 words of the last valid LAT address.

Refer to Figure 77 for the IC options bit layout, to Figure 78 for an IC block diagram, and to Figure 79 for the IC option terms.

Figure 80 is a block diagram of the memory-to-instruction buffers for path 1, and Figure 81 is a block diagram of the memory-to-instruction buffers for path 2. Figure 82 is a block diagram of the common memory path code 1 fanouts, and Figure 83 is a block diagram of the common memory path code 2 fanouts.



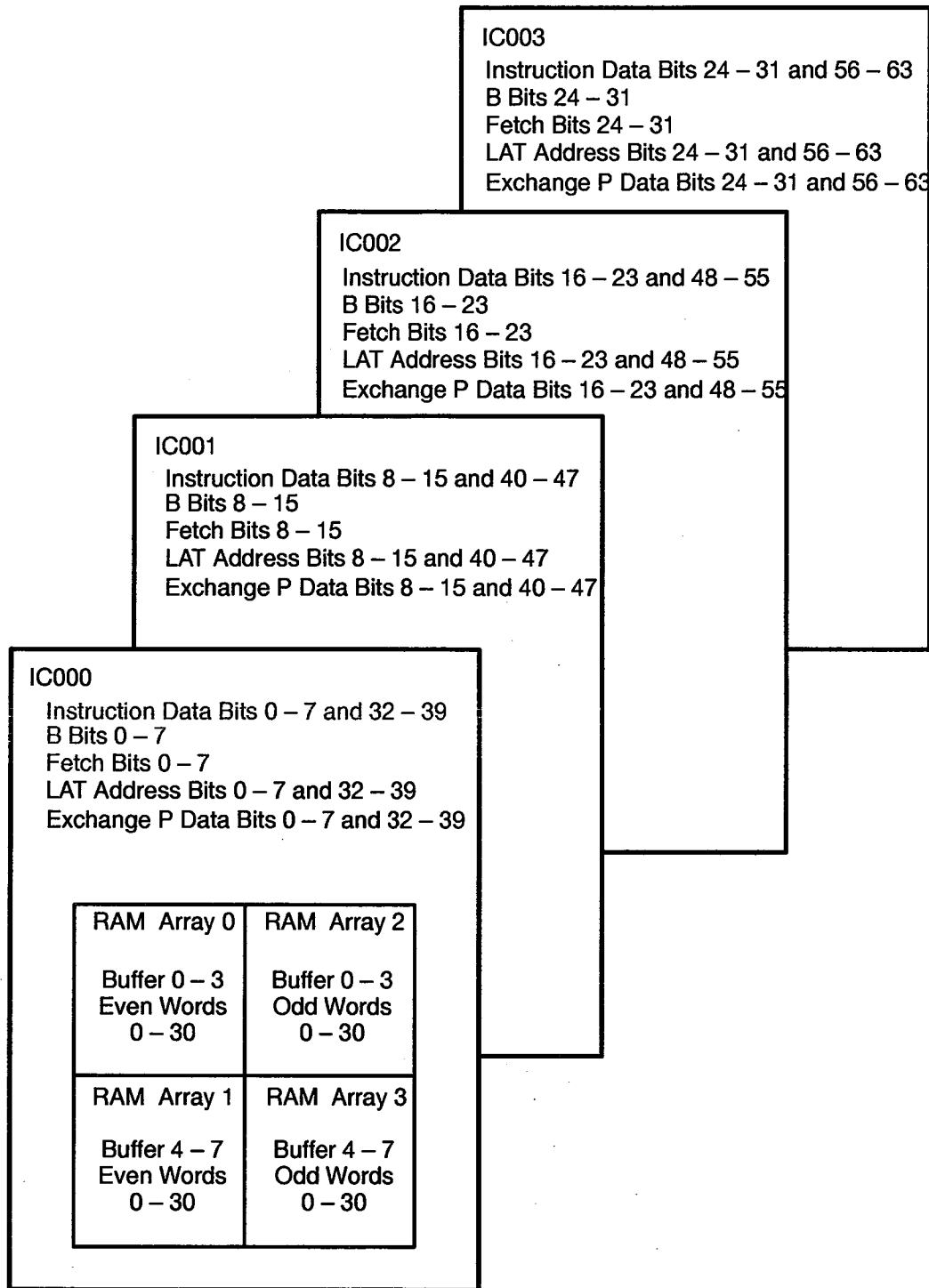


Figure 77. IC Options Bit Layout

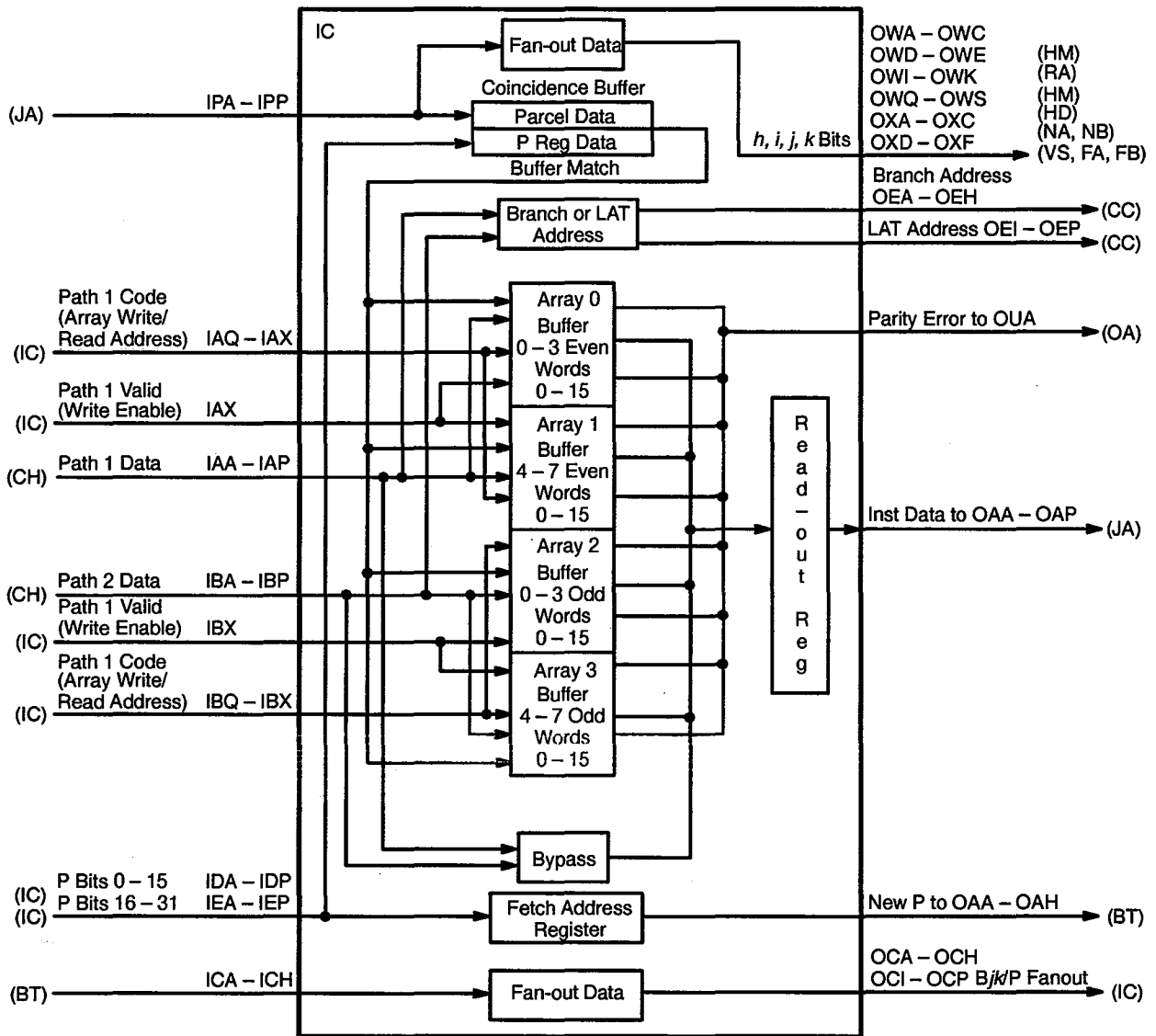


Figure 78. IC Block Diagram

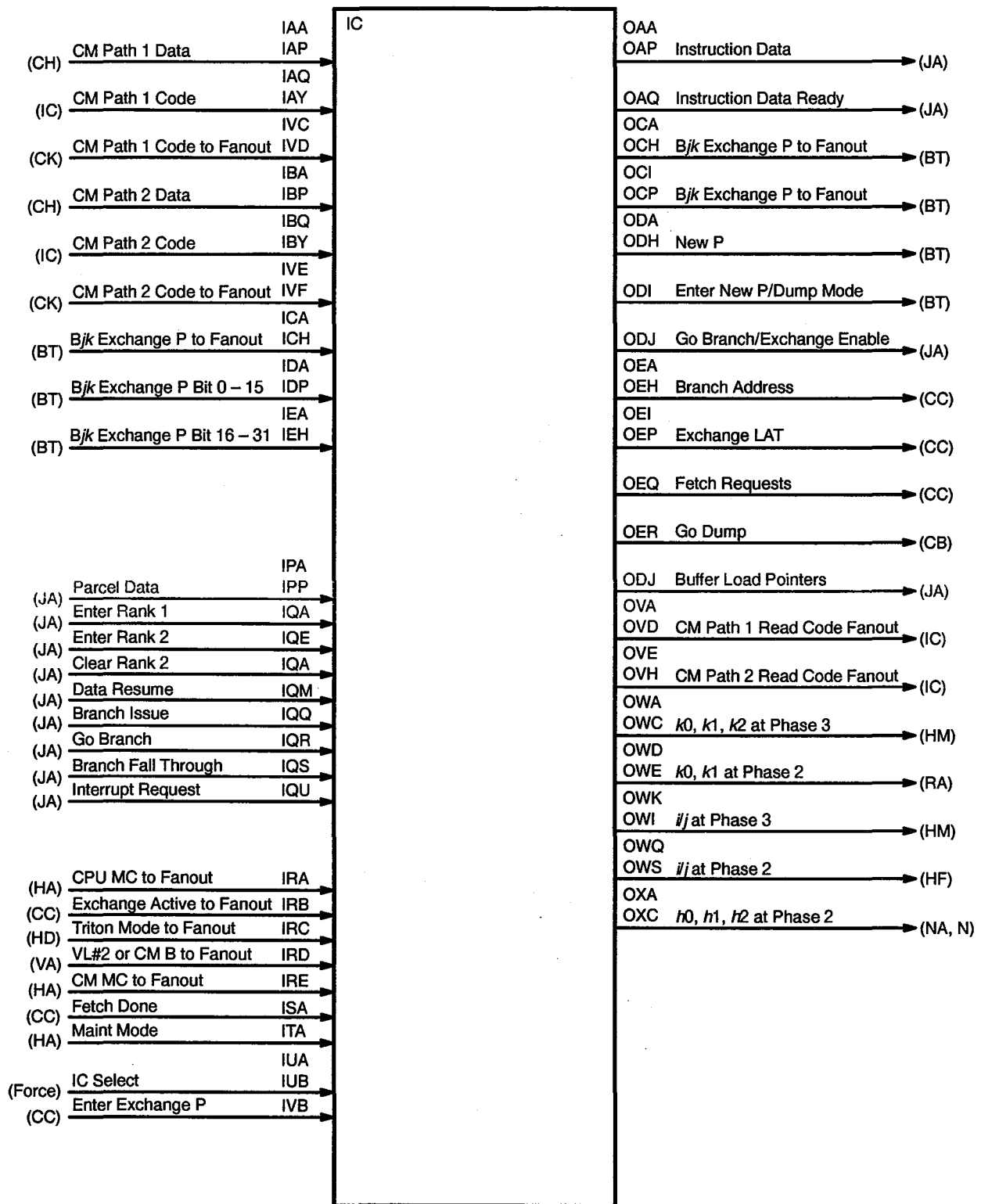


Figure 79. IC Option Terms

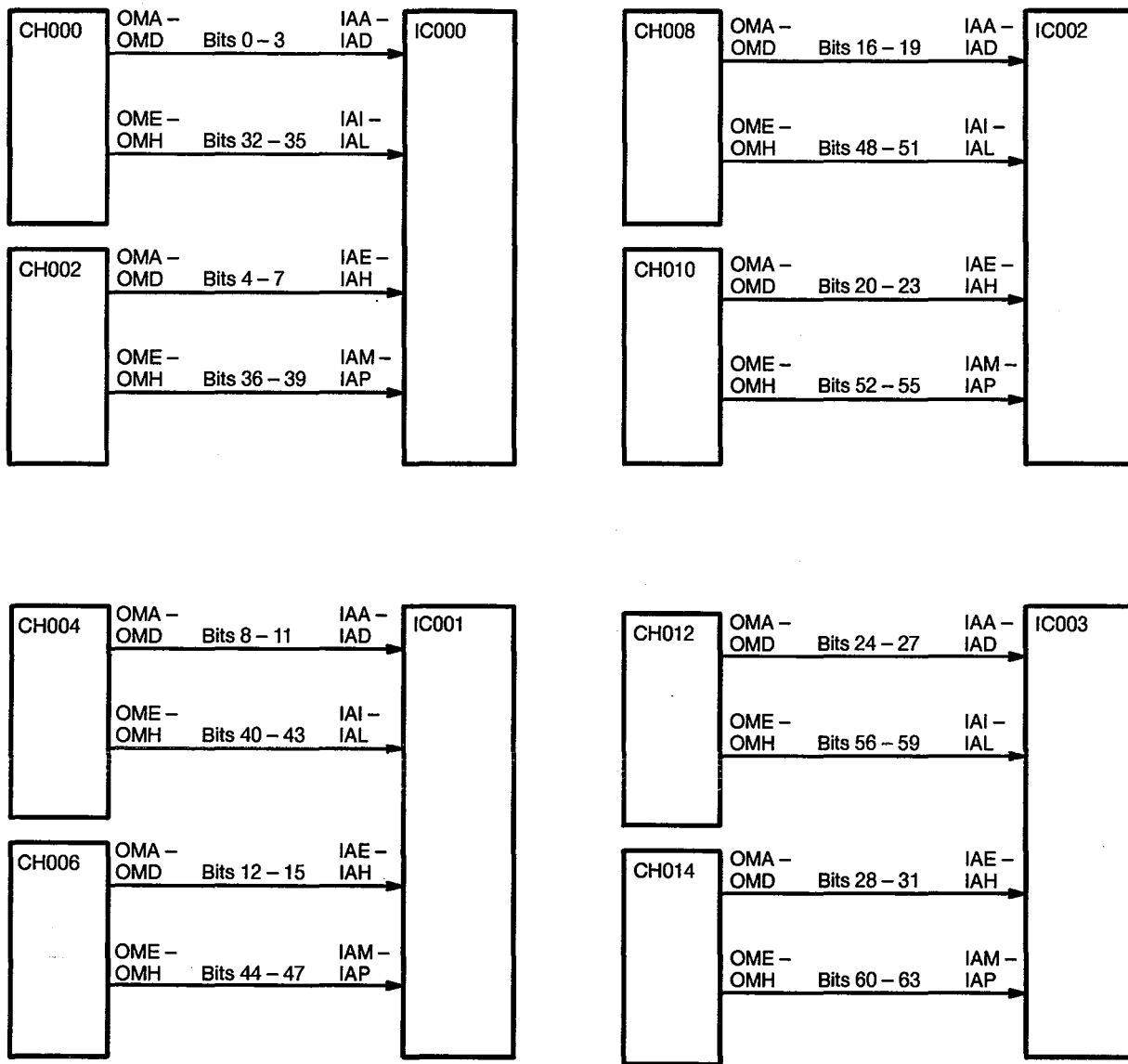


Figure 80. Memory-to-instruction Buffers (Path 1)

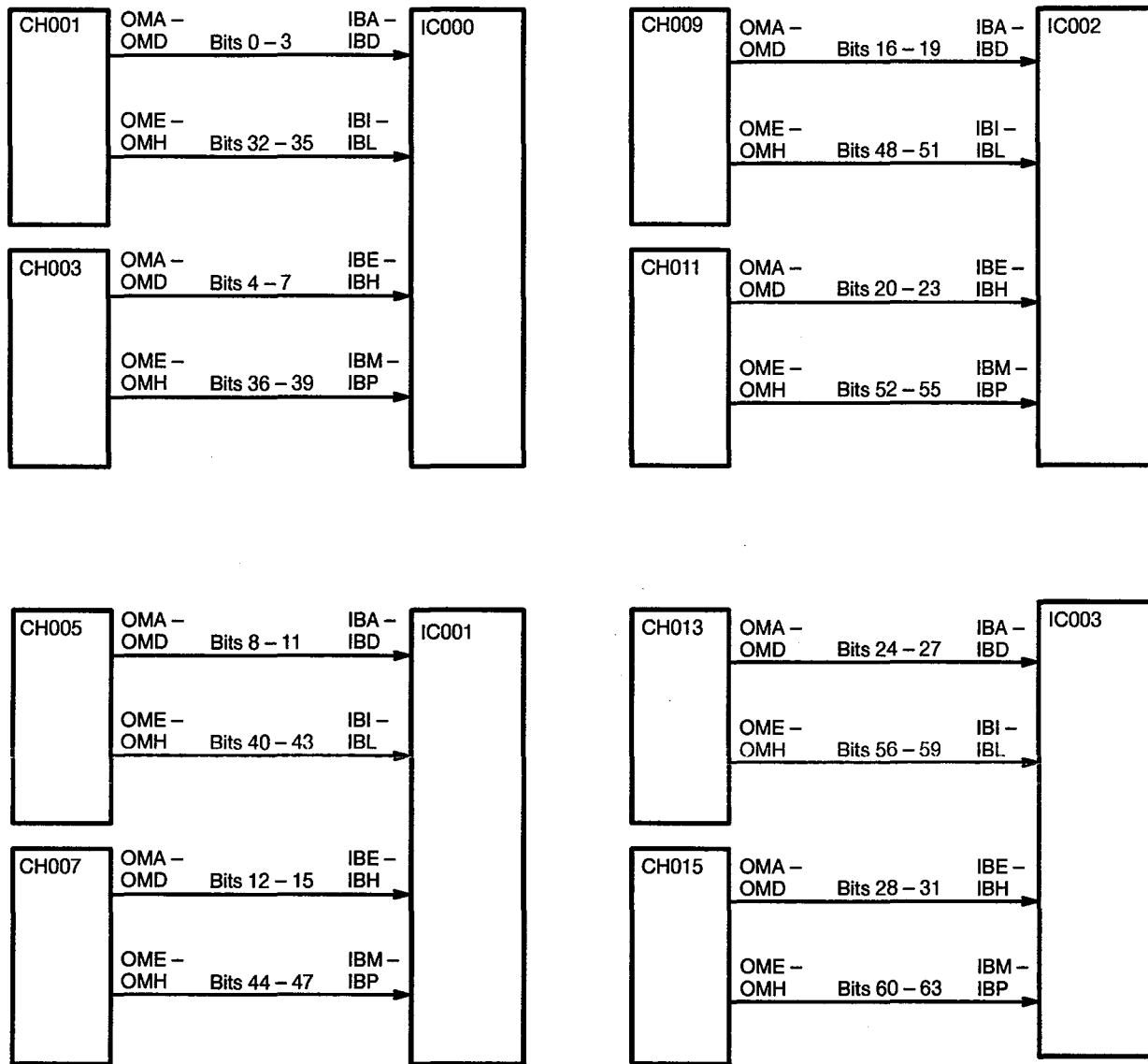


Figure 81. Memory-to-instruction Buffers (Path 2)

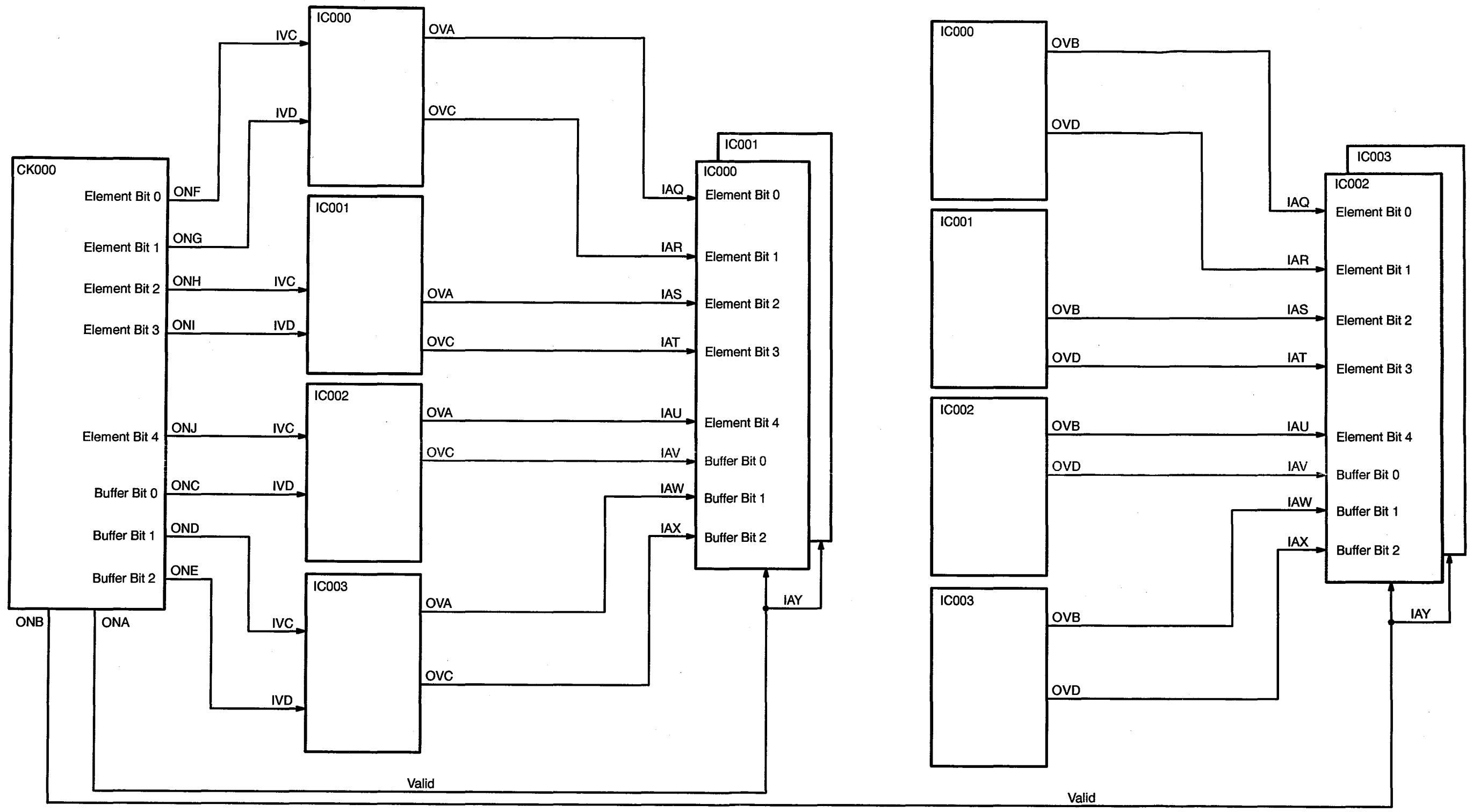


Figure 82. Common Memory Path Code 1 Fanouts

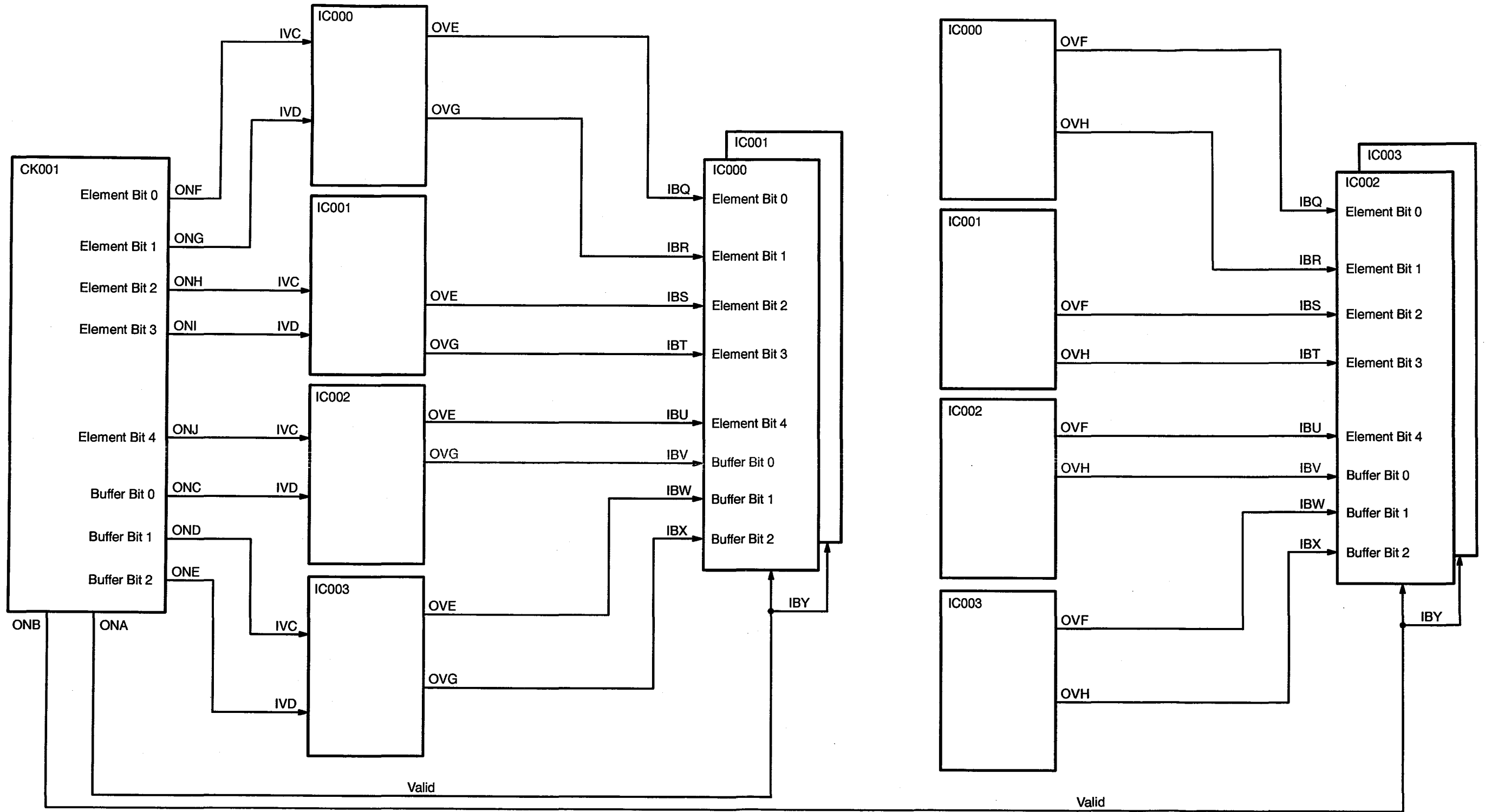


Figure 83. Common Memory Path Code 2 Fanouts

# INSTRUCTION ISSUE

A CRAY T90 series computer system uses a process called instruction issue to introduce instructions into the central processing unit (CPU).

The first instruction parcel is read from one of eight instruction buffers (IBs) and sent to the next instruction parcel (NIP) register where it is partially decoded to determine whether it is a 1-, 3- or 4-parcel instruction.

Refer to Figure 84 for an instruction issue block diagram. The program address (P) register points to the next parcel to be read out of the instruction buffer. If it is a 1-parcel instruction, the NIP moves to the current instruction parcel (CIP), the parcel from the instruction buffer moves to NIP, and P is incremented by 1. If it is a 3-parcel instruction, as NIP moves to CIP, the second parcel moves into LIP0, the third parcel moves into LIP1, and P is incremented by 3. If it is a 4-parcel instruction, as the first parcel moves from NIP to CIP, the second and third parcels move to LIP0 and LIP1. Then, the fourth parcel goes to NIP and then to CIP as the other three parcels are leaving. In the next clock period, the fourth parcel leaves CIP, and P is incremented by 4.

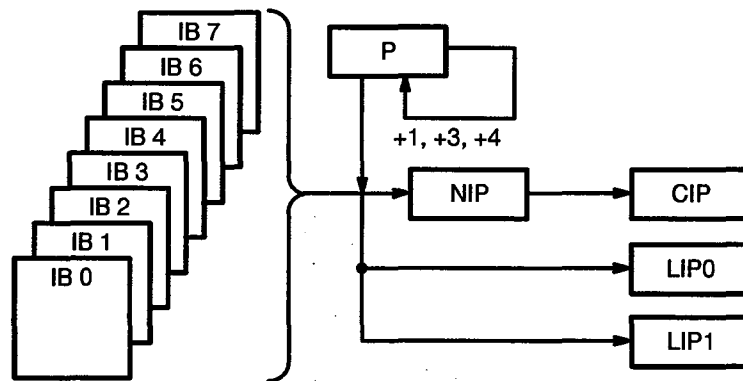


Figure 84. Instruction Issue Block Diagram



## Instruction Formats

There are three instruction formats: 1-, 3-, or 4- parcel instructions. The first parcel always contains the operation code. The operation code is pre-decoded in NIP to determine whether it is an exit instruction (000000 or 004000) or a 1-, 3-, or 4- parcel instruction.

### One-parcel Instructions

The *gh* portion generally is the operation code, although some instructions also use the *i*, *j*, or *k* fields. The *i* field is usually the result designator, and the *jk* portions are generally operand register designators. Some instructions use the *i* field or bit 2 of the *j* field to provide additional bits for the operation code.

Some 1-parcel instructions are part of the extended instruction set (EIS) and perform different operations when immediately preceded by the EIS parcel (005400).

Figure 85 shows the format of a 1-parcel instruction.

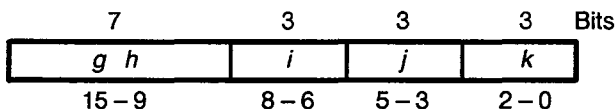


Figure 85. Format for a 1-parcel Instruction

### Three-parcel Instructions

The 3-parcel instruction is used in both Triton mode and C90 mode. The *nm* fields hold the 32-bit address or constant value. Refer to Figure 86 for an illustration of a 3-parcel instruction format.

**NOTE:** The *n* portion holds the most significant bits, and the *m* portion holds the least significant bits.

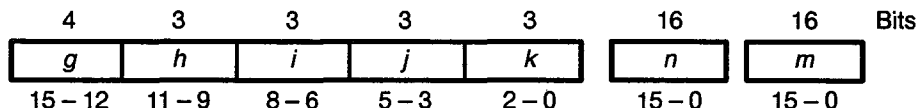


Figure 86. Format for a 3-parcel Instruction

## Four-parcel Instructions

Four-parcel instructions are used exclusively in Triton mode. The instruction field mnemonic *pmn* represents a 48-bit field with the *p* field being the most significant parcel. Refer to Figure 87 for an illustration of a 4-parcel instruction format.

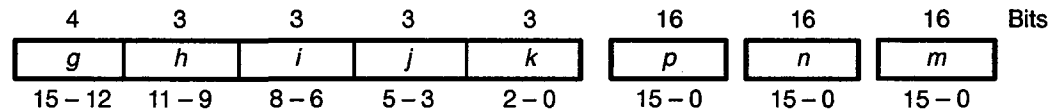


Figure 87. Format for a 4-parcel Instruction

Four-parcel instructions are used for A and S register memory references that use extended addressing. The *h* field selects an A register to be used as an address index. The *i* field designates an A or S register to be used as the source or destination of the data. For read references, *j* field bit 1 disables or enables cache bypass. Bit 2 of the *j* field must be set to a 1 to indicate a 4-parcel instruction. The *k* field is not used.

## Triton-mode Instructions

---

Triton mode is active when the Triton mode bit (TRI) is set in the exchange package. Some instructions execute correctly only in Triton mode. If a Triton mode instruction is executed while the machine is in C90 mode, the results are undefined. Refer to the instruction set for Triton-mode only instructions.

## Instruction Decode

---

After the instruction parcel is in NIP, it is pre-decoded to determine its size. If it is a 1-parcel instruction, it moves to CIP for further decoding to determine which registers, functional units, and memory ports are required.

## P Register

---

The P register is 32 bits wide and resides on the BT0 and BT1 options. The P register points to the relative memory address of the next instruction to be read out of the instruction buffer read-out register and sent to either NIP or LIP0. The lower 2 bits (bits -1 and -2) point to the parcel, and the upper 30 bits (bits 8 through 29) point to the word address. There are three ways to load the P register:

- Multiplex 8 bits at a time during an exchange sequence
- Load from *Bjk* as a result of a *005ijk* instruction
- Load from the *ijk* or *nm* fields of a *006ijk*, *007ijk*, or *01xjk* instruction

Every time a parcel issues, the JA option sends an **Advance P** signal to the BT options, advancing the P register by 1.

## Coincidence

---

A condition called *coincidence* exists if the next parcel needed is in one of the eight instruction buffers. A coincidence check compares the upper 25 bits of the P register to the 25-bit buffer address (A) register as well as determines whether the buffer valid bit is set. All 25 bits must match, and the buffer valid bit must be set in order for a coincidence condition to exist. If there is no coincidence, a fetch operation is initiated.

Coincidence is checked only on branch instructions to determine if the next instruction will be in the stack.

## Reading the Instruction Buffer

---

When a buffer read occurs, both the even and odd words are read out of the buffer to a read-out register. The content of the P register on the BT options directs one of these words to NIP or LIP for decoding.

## JA Option

There are two JA options on the CP module; they provide the issue control signals for the processor. These options receive the instruction word from the IC options, select and decode the correct parcels, and provide control to the rest of the CPU. The JA option also has all the resource reservations and holds issue if a resource is busy. The JA options are responsible for the functions described in the following subsections.

### Parcel Data Distribution

The JA option transmits parcel data to the AR, AS, AT, AU, BT, and VA options and alters the *j* field going to the AR, AS, AT, and AU options for certain instruction types. This occurs on the following instructions:

- 10*h*, 11*h*, 12*h*, 13*h*; the *A<sub>j</sub>* becomes the *A<sub>h</sub>* field
- 0013*j*0; the *A<sub>i</sub>* field becomes the *A<sub>j</sub>* field

The JA option also transmits a read-out pointer code to the A and S registers; the read-out pointer code selects the read-out path. Refer to Table 29 for a list of these codes.

Table 29. Read-out Path Codes

Code	Instruction	Description
00	075, 13 <i>h</i>	<i>S<sub>i</sub></i> to BT path
01	034, 036, 025, 11 <i>h</i>	<i>A<sub>i</sub></i> to BT path
11	035, 037	<i>A<sub>i</sub></i> to BT path
00	0013 <i>j</i> 0, 027 <i>ij</i> 2/3, 027 <i>ij</i> 6/7	<i>A<sub>i</sub></i> to SR path
01	073 <i>ij</i> 2, 073 <i>ij</i> 3, 073 <i>ij</i> 5, 073 <i>ij</i> 6	<i>S<sub>i</sub></i> to SR path
10	0010 <i>jk</i> , 0011 <i>jk</i>	<i>A<sub>k</sub></i> to SR path
11	0014 <i>j</i> 0, 0014 <i>j</i> 4	<i>S<sub>j</sub></i> to SR path
00	057, 0030 <i>j</i> 0/1, 026 <i>ij</i> 0/1, 027 <i>ij</i> 0	<i>S<sub>j</sub></i> to shift path
11	052 – 056	<i>S<sub>i</sub></i> to shift path
00		<i>S<sub>j</sub></i> to vector pipe 0
01	176	A0 to vector pipe 0
10	034, 036	A0 to vector pipe 0
11	035, 037, 177	A0 to vector pipe 0
00		<i>S<sub>j</sub></i> to vector pipe 1

Table 29. Read-out Path Codes (continued)

Code	Instruction	Description
01	176	Ak to vector pipe 1
10	034, 036	Ai to vector pipe 1
11	035, 037, 177	A0 to vector pipe 1
00	10h, 12h, 13h, 0017jk	Ah (A) to CM port B/E
01	00200k	Ak to CM port B/E
10	11h	Ah (A) to CM port B/E
11	177	Ak to CM port B/E

### A/S/V/B/T Register Requests

The JA option checks for register conflicts and receives a register release signal from the shared resource control and from common memory for the A and S registers. The JA option also receives a vector read/write (R/W) release for V registers and a B/T read/write release. The JA option also transmits A and S register entry codes. These codes, along with the *ghijk* field, the instruction, and the 2-bit register read-out code are used by the A and S registers to define the instruction to be performed and to reserve the needed path.

### Functional Unit Requests

The JA option checks for functional unit conflicts in the following functional units:

- Logical #1: 140 – 147 / 175
- Logical #2: 140 – 145 if Logical #1 busy / Logical #2 enabled
- Vector Mask: 146 – 147 / 175 / 070ij1 / EIS 153ij0,1
- Vector Shift: 150 – 153
- Vector Add: 154 – 157
- Floating Multiply: 160 – 167
- Floating Add: 17 – 173
- Reciprocal (V pop, parity, leading zero, iota: 174ij(0 – 3) / 070ij1
- Matrix Multiply: 174ij(4 – 7) / 070ij(6 – 7)

## Constant Data Requests

The JA option checks for constant data present on multiple-parcel instructions such as jumps, branches, and instructions using the *pmn* fields. Each JA option handles 32 bits of the constant data distribution. JA0 transmits data to the AR, AS, and CD options via the A series options, and JA1 transmits data to the AT, AU, and CD options via the A series options. JA0 also provides the *jk* data on the constant path when needed.

## EIS (Extended Instruction Set) Requests

The JA option issues 005400 as a normal instruction; however, the next parcel is decoded using the extended instruction set. If an EIS instruction is issued without the 005400 preceding it, the instruction issues and performs its normal function. For example:

044ijk Transmit logical product of (*Sj*) and (*Sk*) to *Si*

044ijk In EIS mode, the same instruction transmits logical product of (*Aj*) and (*Ak*) to *Ai*

## Common Memory Requests

The JA options receive the following external common memory control signals:

- **Release Port A**
- **Release Port B**
- **Release Port C**
- **Bidirectional Mode:** (Mode = 1) enable block reads and writes at the same time
- **Common Memory Quiet:** This signal indicates that all memory activity in the CPU has been completed. It requires that all ports are quiet, conflict logic is quiet, memory sections are quiet, and all read and write operations are complete.
- **Hold Common Memory Issue:** No more references can issue
- **Cache Miss In Progress:** Indicates a cache miss is pending

- **Read Quiet:** Read references have cleared all conflict checks
- **Write Quiet:** Write references have cleared all conflict checks
- **Exchange Active:** Indicates an exchange has not completed

## Shared Resource Requests

The JA options receive the following external signals, which control the shared resource path, from the HD option:

- **A/S Register Shared Resource Release:** Releases a specific A or S register (0 – 7) path
- **Release Shared Resource:** Used in combination with Go Semaphore Branch to cause issue to resume or P to advance
- **Go Semaphore Branch:** Signals that the conditions of a semaphore branch have been satisfied

## Branch Requests

The JA options check the branch test conditions to determine whether the condition is met; if it is, the JA option issues a **Go Branch** signal to the IC options.

## Exchange Requests

The JA options perform the following actions during an exchange sequence:

- 000000 (error exit) issues. Issue stops, P advances
- 0040jk (exit k) issues. Issue stops, P stops
- The shared path is released. The state of **Go Semaphore Branch** determines whether P advances on a 0040jk. Two conditions of the 0040jk instruction could occur:
  1. A normal exit occurs and P advances when the shared path is released and **Go Semaphore Branch** is a 0.
  2. An error exit occurs, P will not advance when the shared path is released, and **Go Semaphore Branch** is a 1.

## Interrupt Requests

An interrupt request can be generated in one of three ways:

- A 000000 (error exit) instruction issues
- A 0040*jk* (Exit *k*) instruction issues
- A hardware error condition occurs

Interrupt requests are processed in two phases. In phase 1, the following conditions are checked:

- No multiparcel instructions are in process
- No EIS type waiting for second parcel
- No branch sequence in progress

In phase 2, the following conditions are checked, and then the **Go Exchange** signal is sent to the HD, IC, and CC options.

- No branch sequence in progress
- Shared path available
- All registers available
- Common memory quiet

When a hardware interrupt request occurs, the JA option performs the phase 1 checks and stops issue. If the phase 2 checks are all valid, the JA option sends a **Go Exchange** signal to the IC options. If any of the shared type instructions have issued during this shut-down time, the HD option must release the shared path and the following actions must occur:

- If a 0034 (test and set semaphore) was issued, a **Release** signal and a **Go Branch** signal must be sent before **Go Exchange** can occur.
- If a 000000 (error exit) or a 0040*jk* (exit *jk*) was issued, a release path must occur to clear the JA option control.

Issue will resume when **Go Branch** occurs.

## Control Signal Distribution

The JA option transmits the following control signals:

- **Issue group 0, 1, and 2:** These signals are combined on the BT and VA options to complete the issue signal.
- **Issue:** This signal is transmitted to the AN option for fanout.



- **Enter Vector Length:** This signal is sent to the AR option on the decode of a 00200*k* (*Ak* to VL) instruction.
- **Read Vector Mask:** This signal is sent to the SS option on a 073*i* (0 – 3) 0 (VM0 or VM1 to *Si* or *Ai*) instruction.
- **Enter Vector Mask:** This signal is sent to the SS option on a 0030*j* (0 – 3) (*Si* or *Ai* to VM0 or VM1) instruction.
- **Go Scalar Pop/Parity/Lz:** This signal is sent to the SS option on a 026*ij* (0 – 3) or 027*ij* (0 – 1).
- **Go Scalar Double Shift:** This signal is sent to the SS option on a 056*ijk* Shift (*Si*) and (*Sj*) left *Ak* places to *Si*.
- **Go A Type:** This signal is sent to the SS option when a 005400 (EIS) is issued using A register data.
- **Go Scalar Reciprocal:** This signal is sent to the RA option on a 070*ij*0 instruction.
- **Go Scalar Floating Add:** JA1 sends this signal to the FA option when a 062*ijk* (sum) or 063*ijk* (difference) issues.
- **Go Scalar Floating Multiply:** This signal is sent to the NA and NC options when a 064*ijk* through 067*ijk* instruction issues.
- **Go Address Multiply:** This signal is transmitted to the AR option when a 032*ijk* issues.
- **Common Memory A or S Requests:** This signal is sent to the CD options when a memory load or store issues. JA0 sends out an A register request, and JA1 sends out S register requests.
- **Common Memory A or S Writes:** This signal is sent to the CD options when a memory write 11*hixxpn*m or 13*hixxpn*m issues. JA0 sends out A register write requests, and JA1 sends out S register write requests.
- **CM Port B Enabled:** This signal is sent to the VA option via the JA0 option and to the BT option via the JA1 options to select the vector read ports.
- **Vector Logical #2 Enabled:** JA0 sends this signal to the VA options to select vector logical functional units.

- **Data Resume:** This signal is sent to the instruction stack (IC options) to indicate that the JA can accept another word.
- **Go Exchange:** This signal is sent to the IC options to indicate that an exchange is required. Another copy is sent to the HD option and is used by the HD's to clear the SIE bit (taking I/O interrupt). The Go Exchange signal is also sent to the CC option to signal the CC to start swapping exchange packages in memory.
- **Go Branch:** This signal is sent to the IC options to indicate that a conditional branch has passed the test.
- **Branch Fall Through:** This signal is sent to the IC options to indicate that a conditional branch has failed the test.
- **Branch Issued:** This signal is sent to the IC options to indicate that a branch has issued.
- **Enter Rank 1, Enter Rank 2, or Clear Rank 2:** These three signals are sent to the IC options to move parcel data into or out of the ranks into issue.
- The following signals are transmitted to the performance (HF) monitor to indicate a hold issue condition:
  - **Holding Issue on A Registers**
  - **Holding Issue on S Registers**
  - **Holding Issue on B/T Registers**
  - **Holding Issue on V Registers**
  - **Holding Issue on Common Memory**
  - **Holding Issue on Functional Unit**
  - **Holding Issue on Shared Resources**
- **Advance P:** This signal is sent to the P register (BT options) to advance P by 1 as each parcel is issued.

## Branch Instruction Control

---

The JA options decode and control the execution of branch instructions. When a conditional branch passes or fails a test, it returns either the **Go Branch** control signal or the **Branch Fall Through** control signal to the IC options. Issue is halted until the **Go Branch** signal is received by the IC options. Another signal, **Branch Issued**, is also sent to the ICs when a branch is in progress.

### Conditional Branch Instructions

Conditional branches use instructions 010*ijk* through 017*ijk*. Once the instruction issues, branch control logic examines either the A0 or S0 register for the condition defined by the operation code. If the condition is met, the value of the P register is replaced with the *nm* field, and program flow is passed to the instruction specified by P. If the condition is not met, program flow drops through to the instruction that follows the branch.

Another type of conditional branch instruction for a CRAY T90 series computer system is called test and set branch (0064*jkmn*). If a specified semaphore register equals 0, the bit is made a 1 and the next instruction issues. If the semaphore is a 1, the P register is replaced with the value in the *nm* field.

### Unconditional Branch Instructions

Unconditional branches use instructions 0050*jk* through 007*ijkmn*, and each code operates differently, except that none of them depends on a condition being met before the branch takes place. In other words, they always take the branch in the *ijkm* or *nm* fields.

The jump to B*jk* instruction (0050*jk*) branches to the parcel address specified by the contents of B*jk*. The unconditional jump instruction (006000*mn*) branches to the *nm* field. A new unconditional jump instruction is the branch to the address in *nm* field (006100*mn*). This instruction is a Triton-mode only instruction; if executed in C90 mode, the results are undefined.

The return jump instruction (007000*mn*) jumps to the address in the address field and places P + 3 (the address of the next instruction) into B00. The return jump allows a jump to a subroutine, the last instruction of which must be a 005000 instruction, which is a jump to B00.

Another new jump instruction is the 007100*nm*, which is an indirect jump. The instruction stores the address of the next sequential instruction in the B00 register; then the instruction uses the *nm* field to specify a common memory address. The lower 32 bits of the contents of that address are transferred to the P register, causing program execution to continue at that point. When this instruction executes, the instruction buffers are set invalid.

## Issue Control

---

The first parcel of the instruction leaves NIP and moves into all the CIPs on options HF000, HD000, and HD001. The CIP located on the HF options is responsible for the instructions that affect the exchange package and performance monitor.

The HD option CIP is used for A/S path release and provides A/S *i* designators and shared path release. The JA options determine whether any register or functional unit reservations exist. If not, these options send the **Issue** signal to the HD and HF options and the instruction issues, reserving the appropriate registers and/or functional unit. If resource conflicts do exist, the JA option does not send the **Issue** signal, and the instruction remains in CIP until the conflict is resolved. This is called a hold issue condition.

The JA options are responsible for providing issue control, and checking and making functional unit and path reservations for the following items:

- Vector registers
- Vector functional units
- A/S shared resource control
- Memory ports
- CM path/cache
- A/S register entry codes
- B/T register

The functional units must send a release back to the JA options to indicate that the units are available.

The JA options also send out the *h*, *i*, *j*, and *k* fields to the A/S registers for further instruction decode.

Refer to Figure 88 through NO TAG for related instruction issue block diagrams.

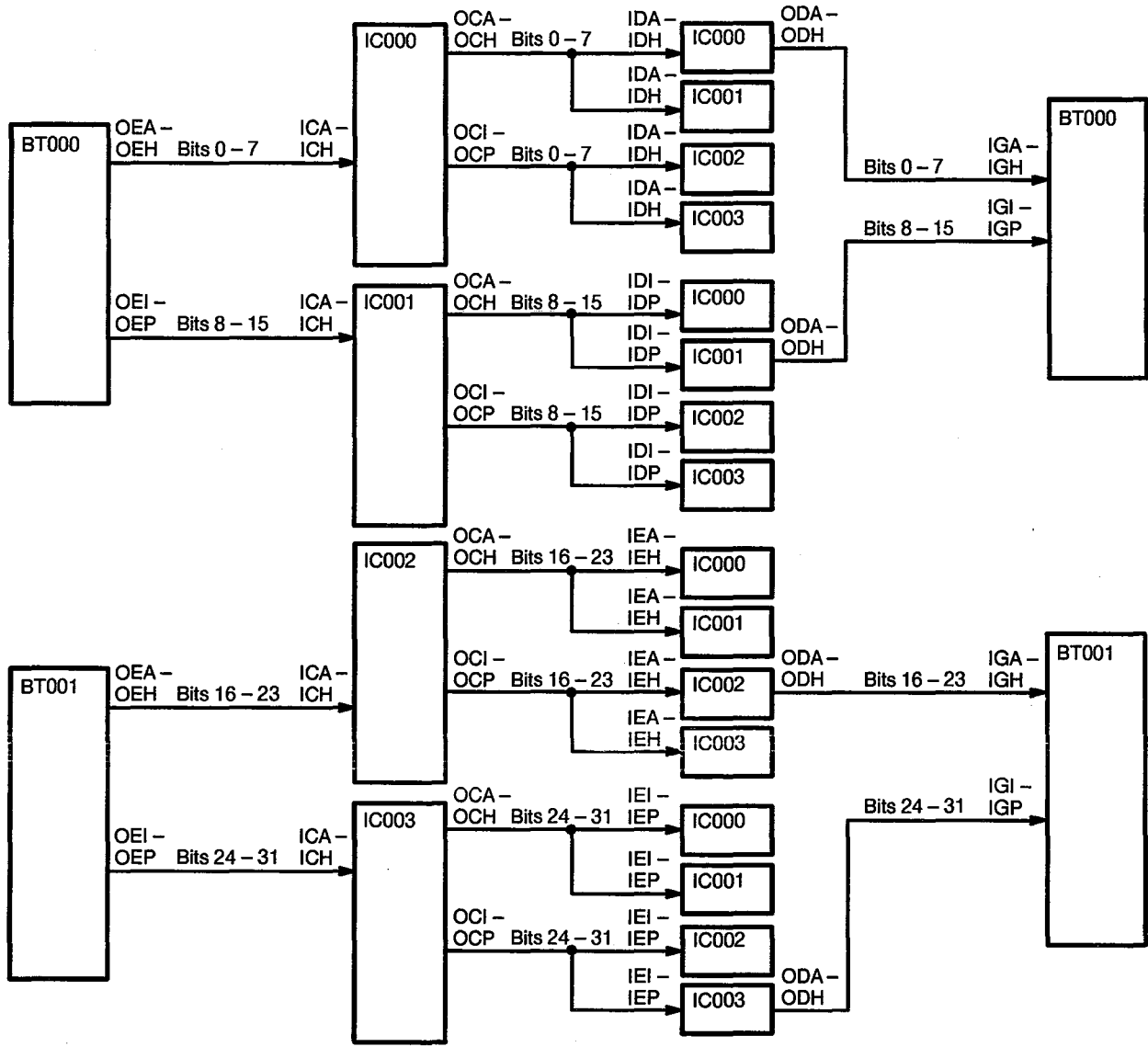


Figure 88. Bjk (Exchange P) Fan-out Bits

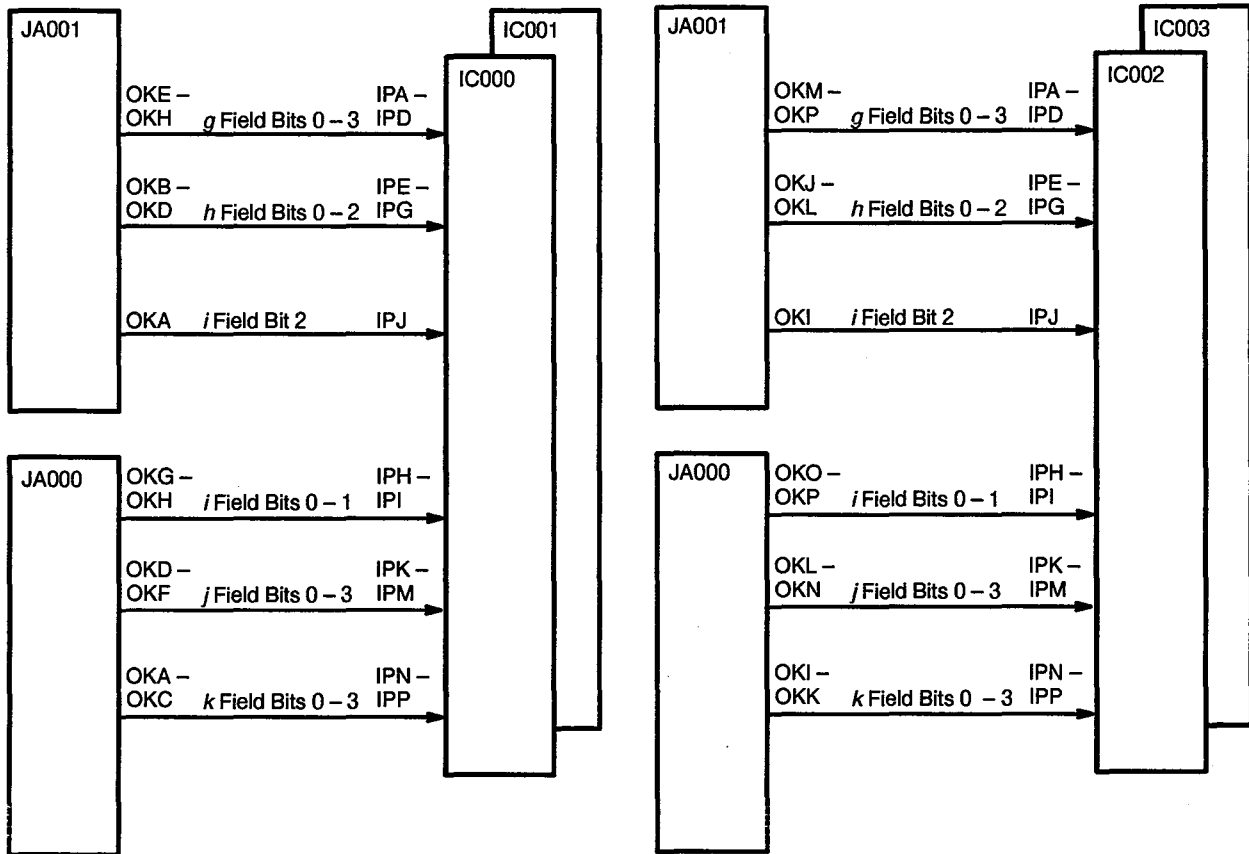


Figure 89. JA-to-IC Parcel Data for Branches

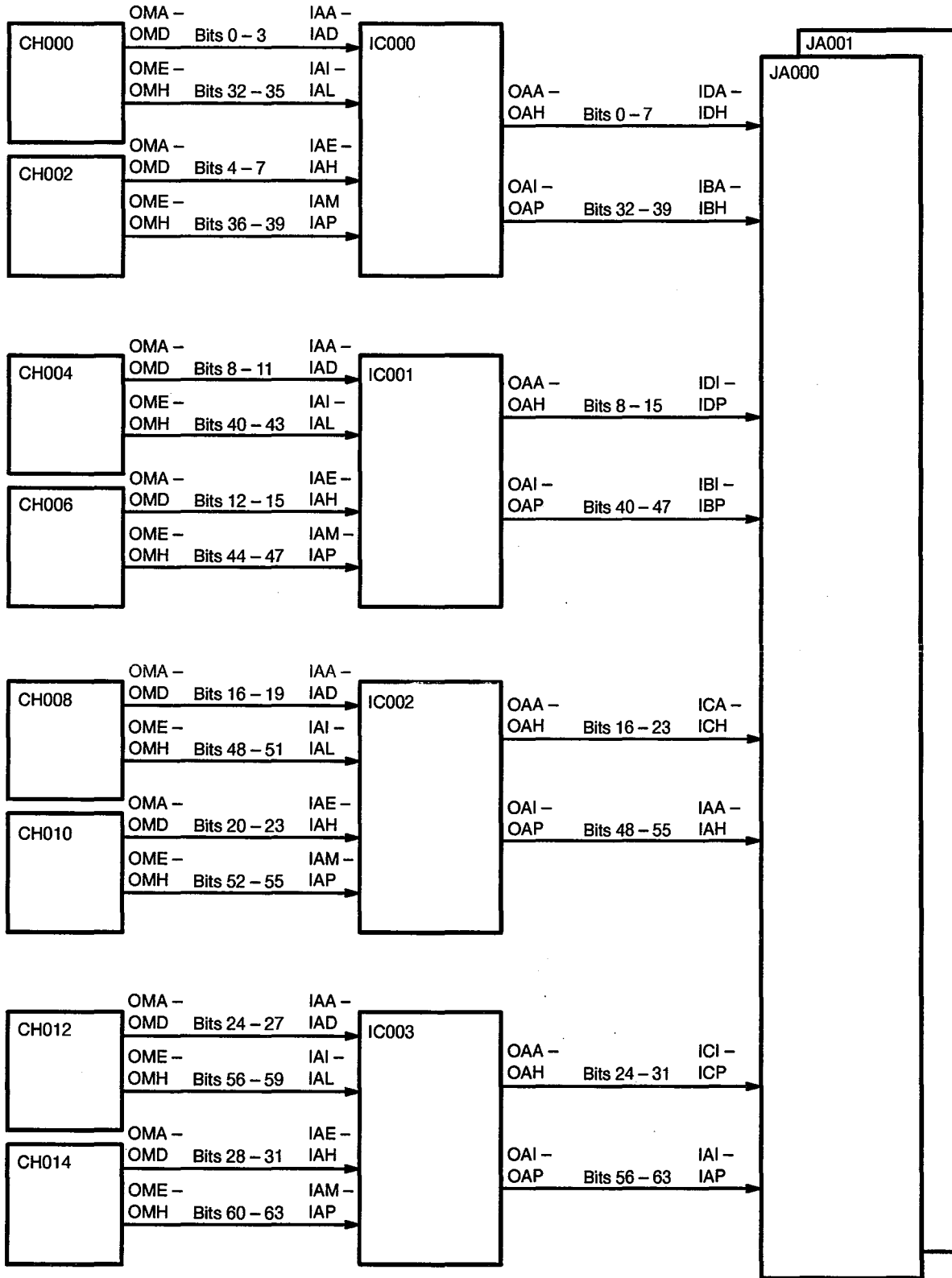


Figure 90. Path 1 CH to IC to JA Option

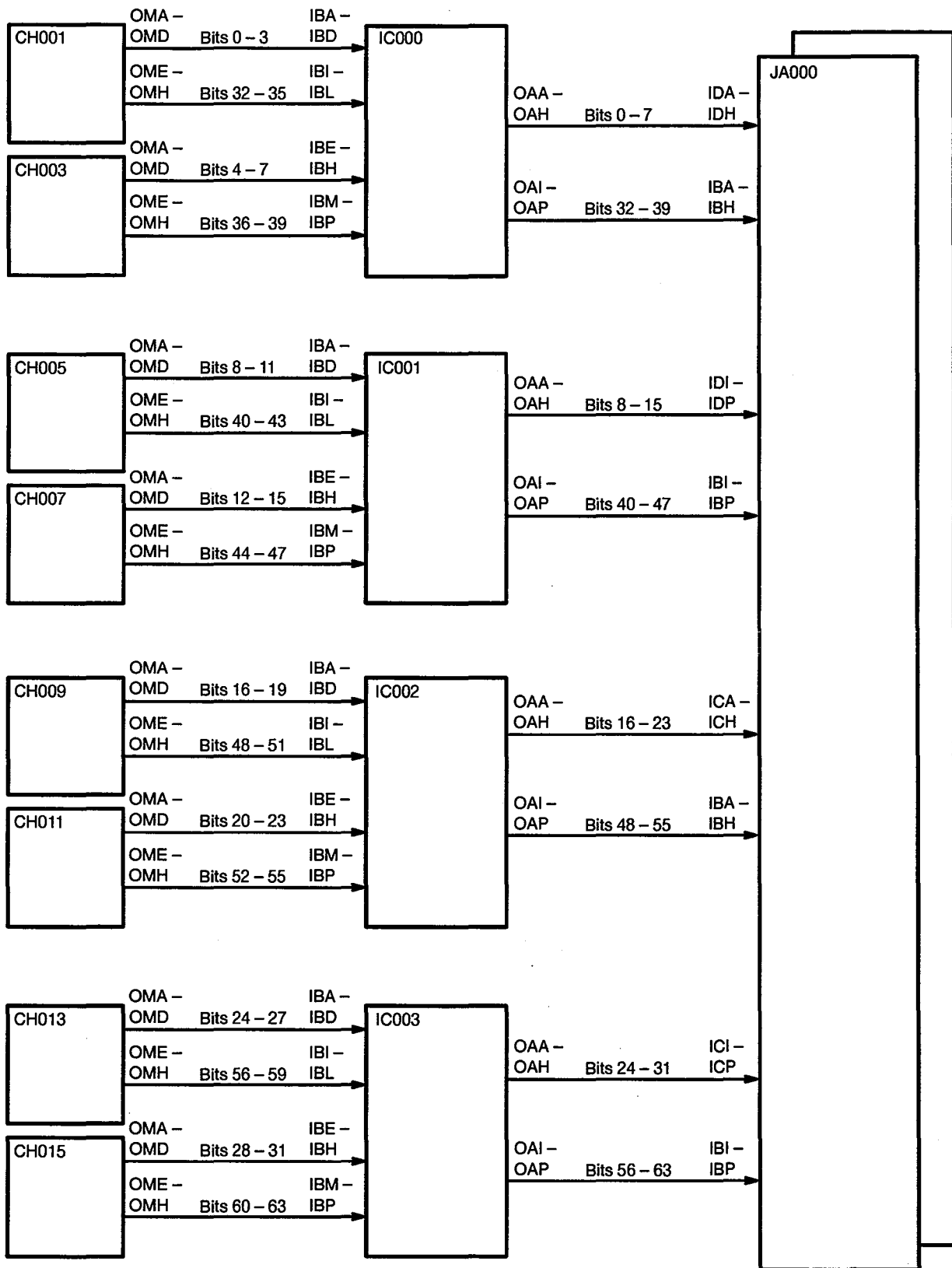
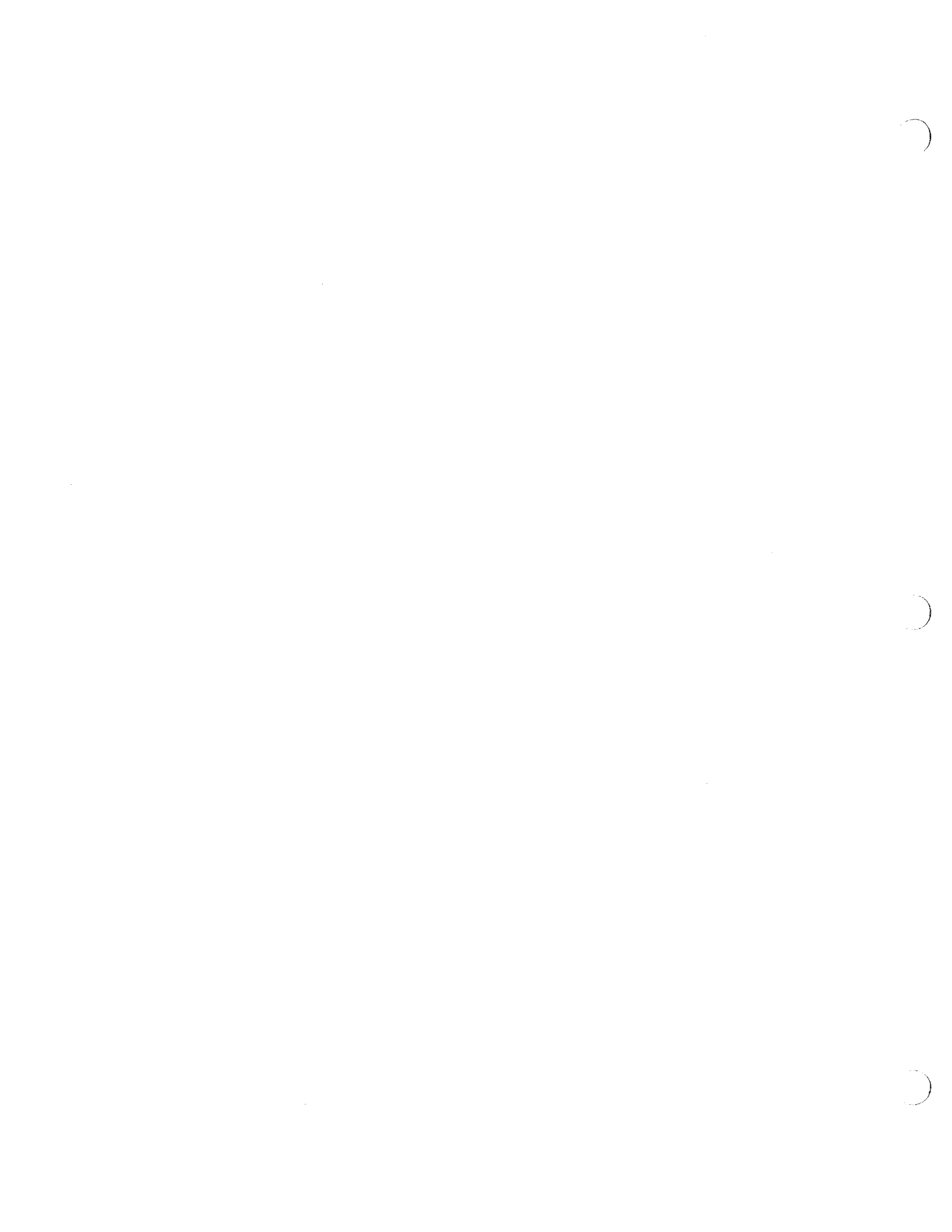


Figure 91. Path 2 CH to IC to JA Option





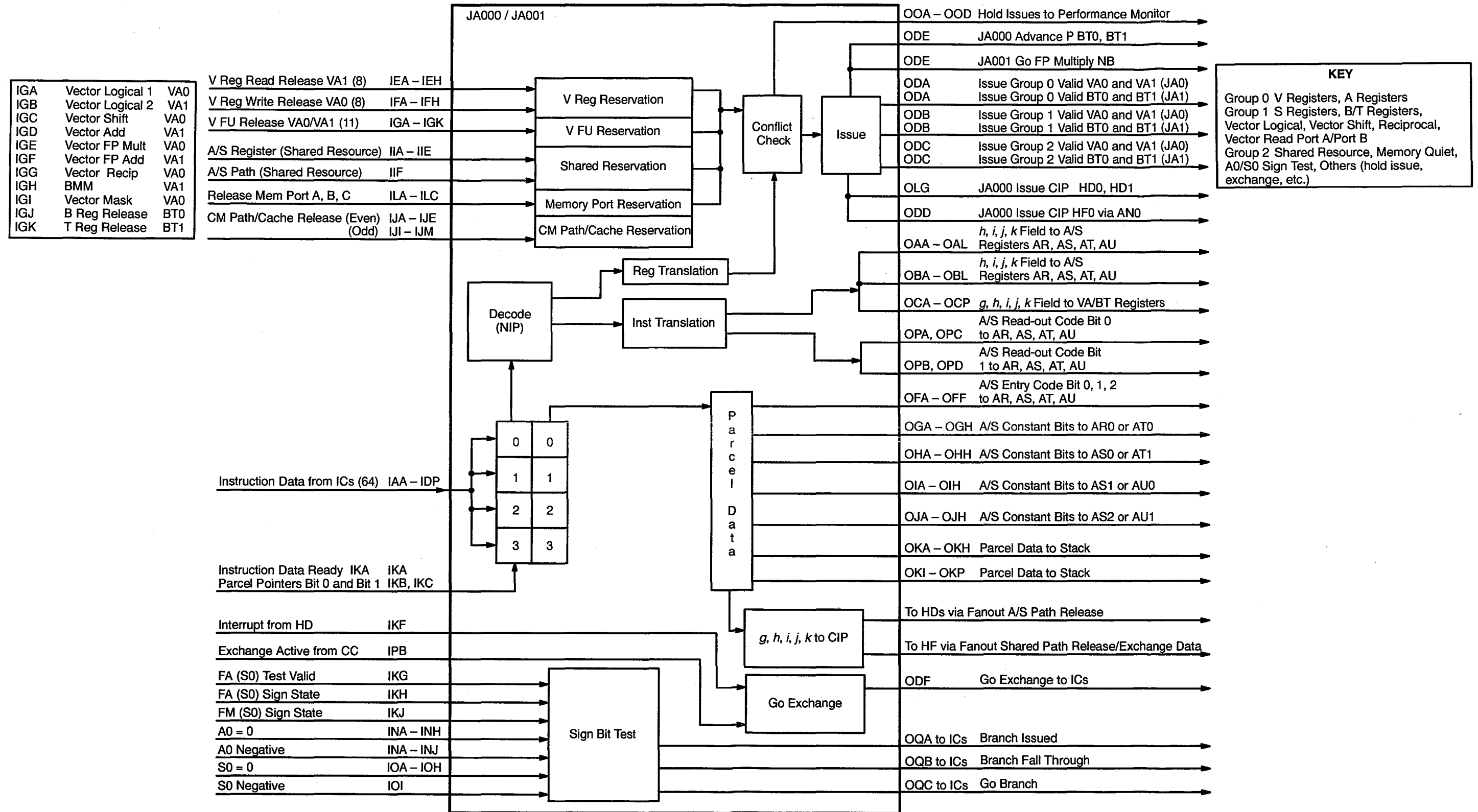


Figure 92. JA Option Block Diagram

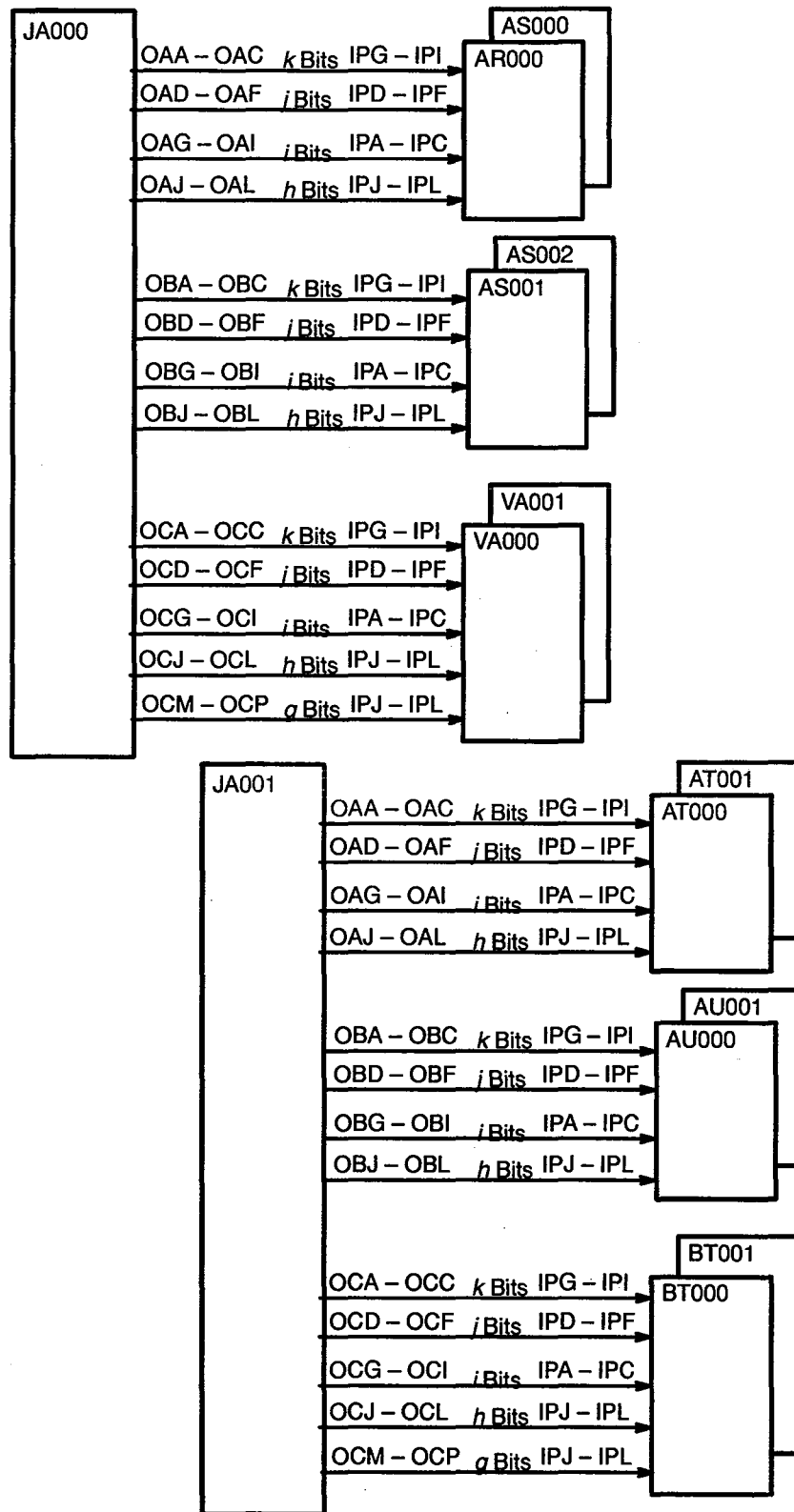


Figure 93. Instruction Data Distribution A/S/B/T Registers

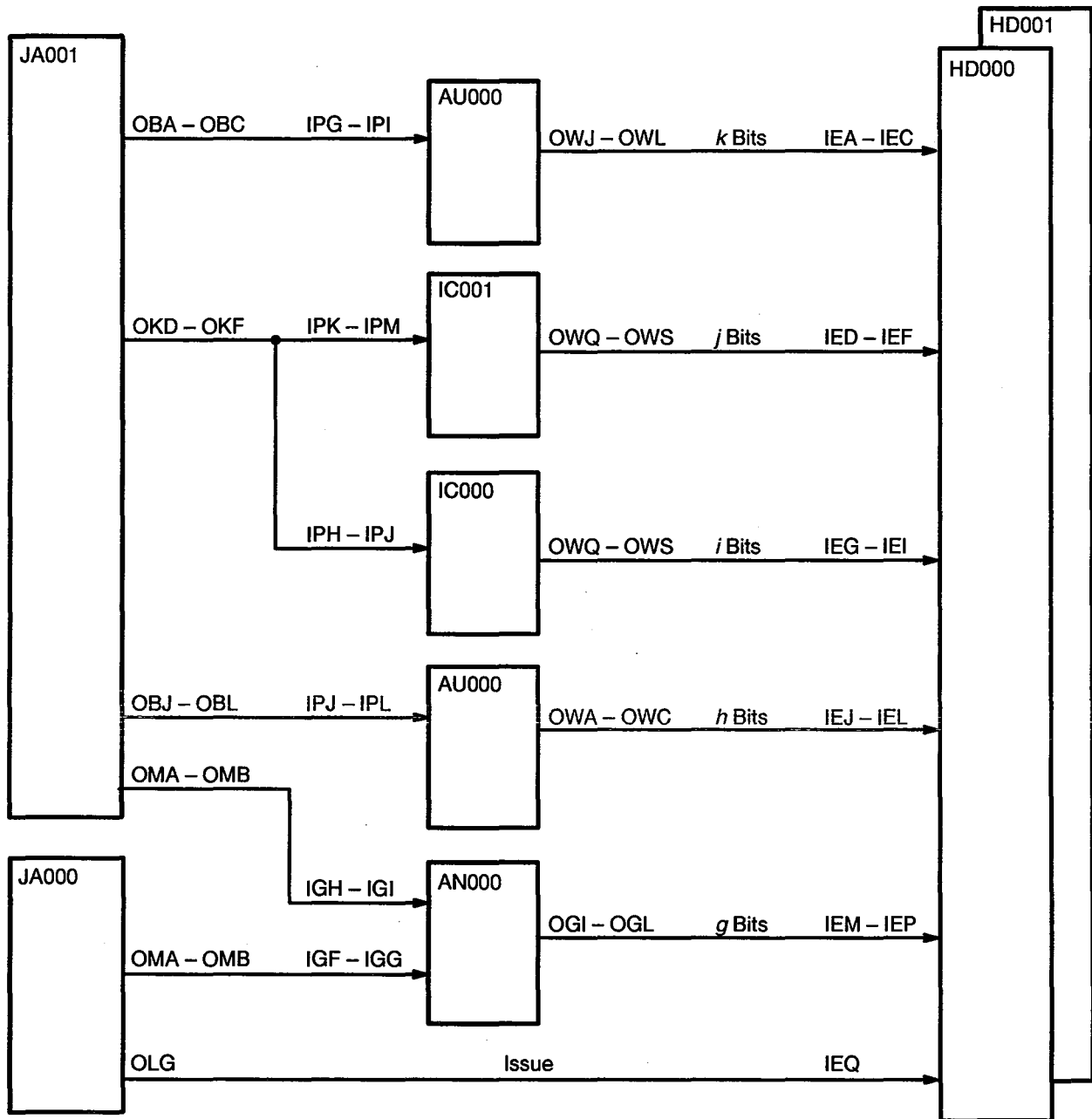


Figure 94. CIP Distribution

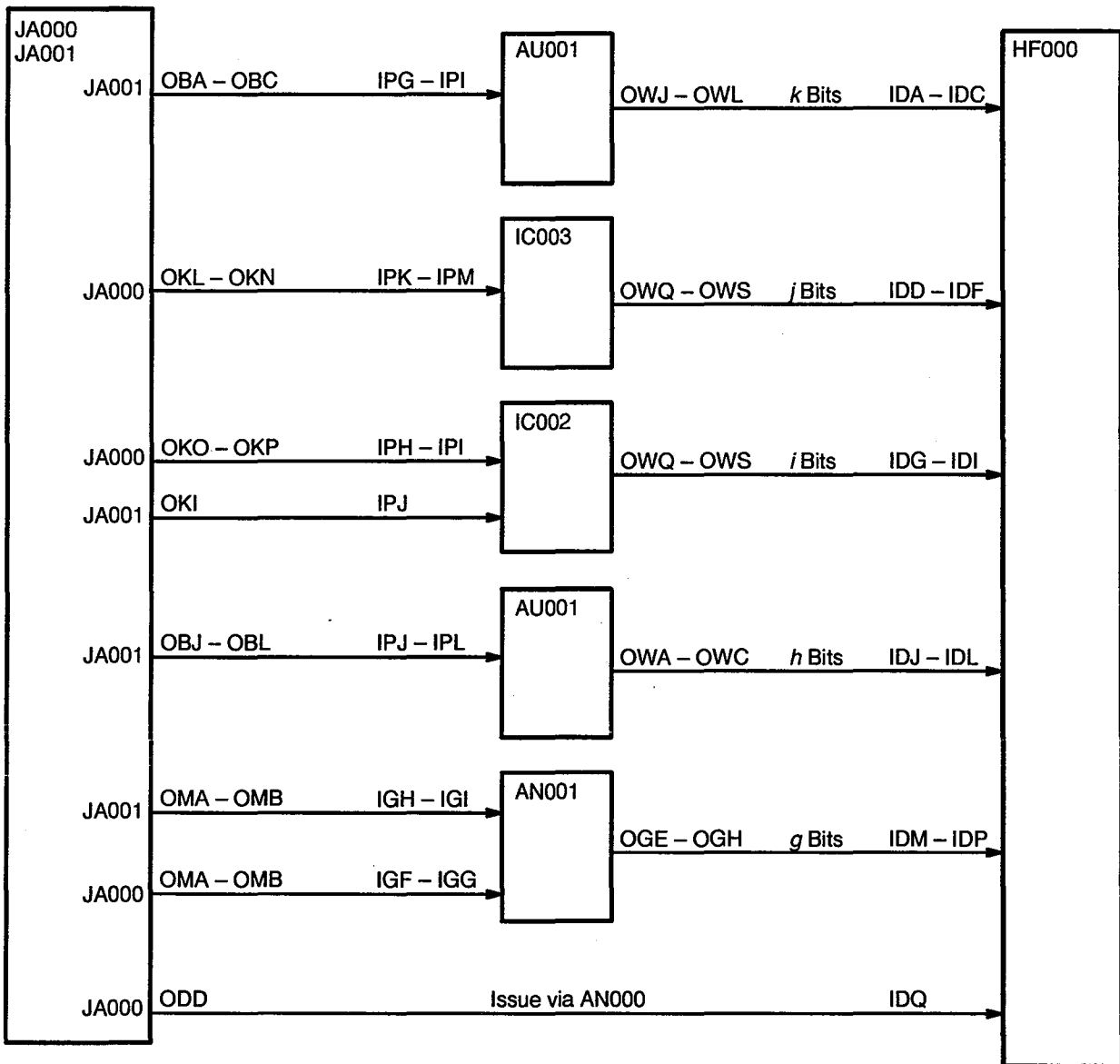


Figure 95. CIP Distribution to HF Option

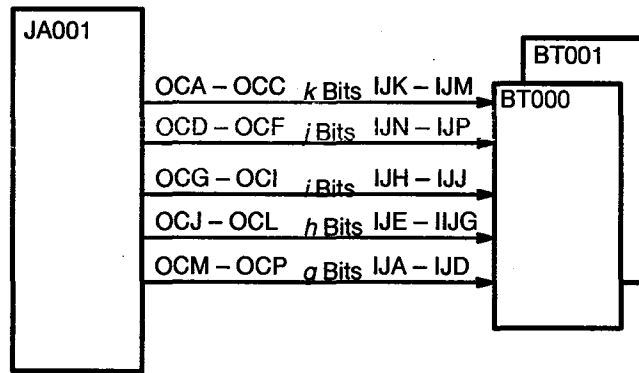
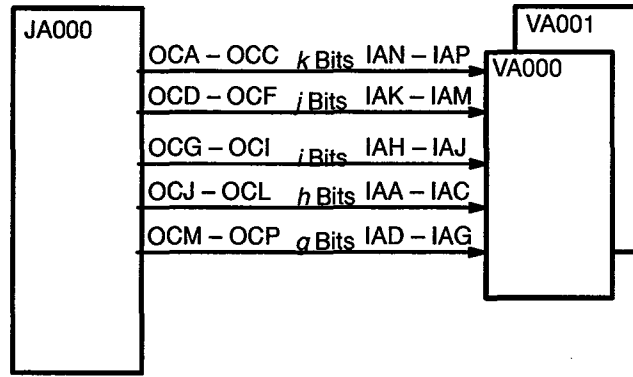


Figure 96. Instruction Data Distribution to VA and BT Options

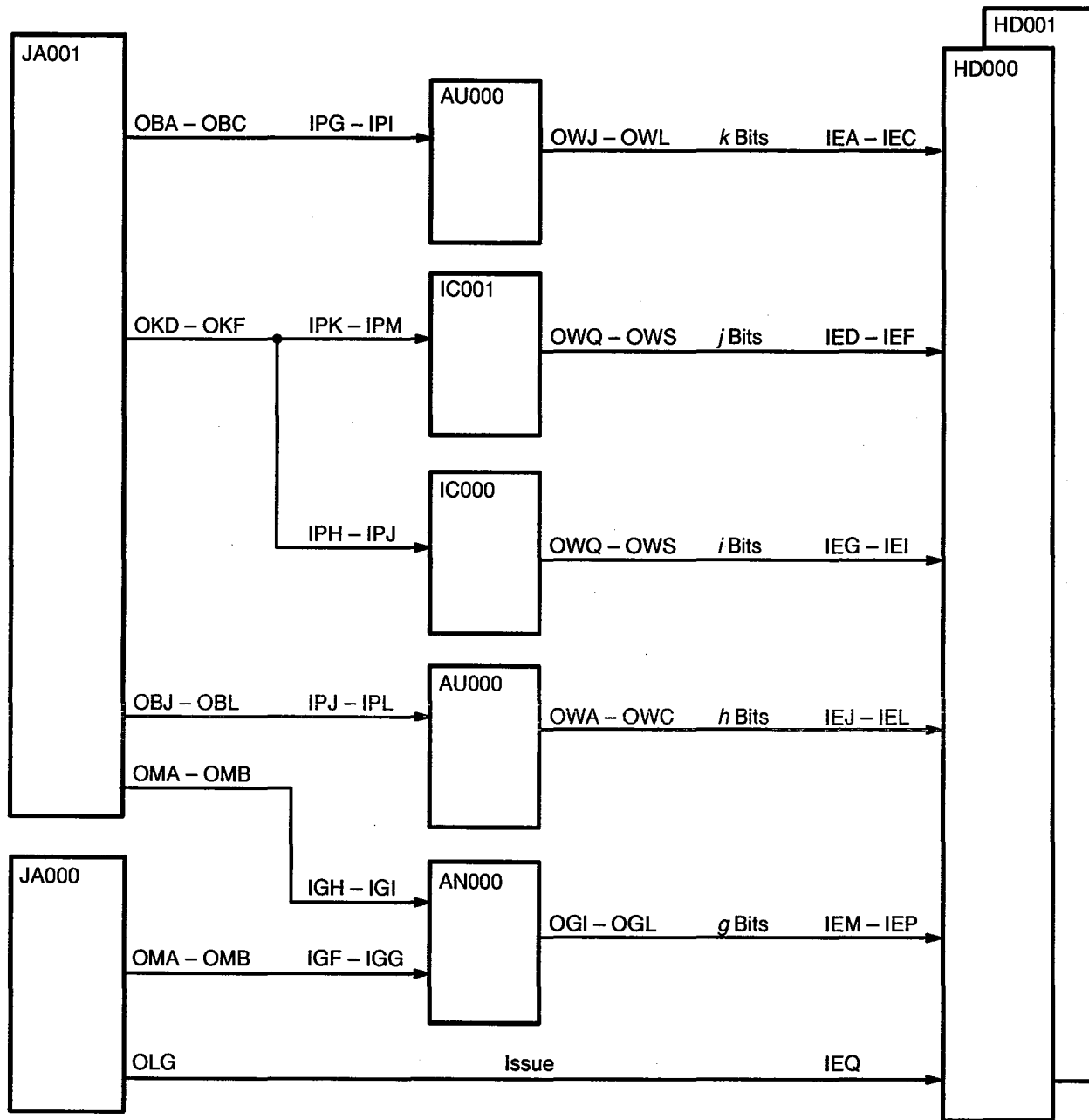


Figure 97. CIP Distribution

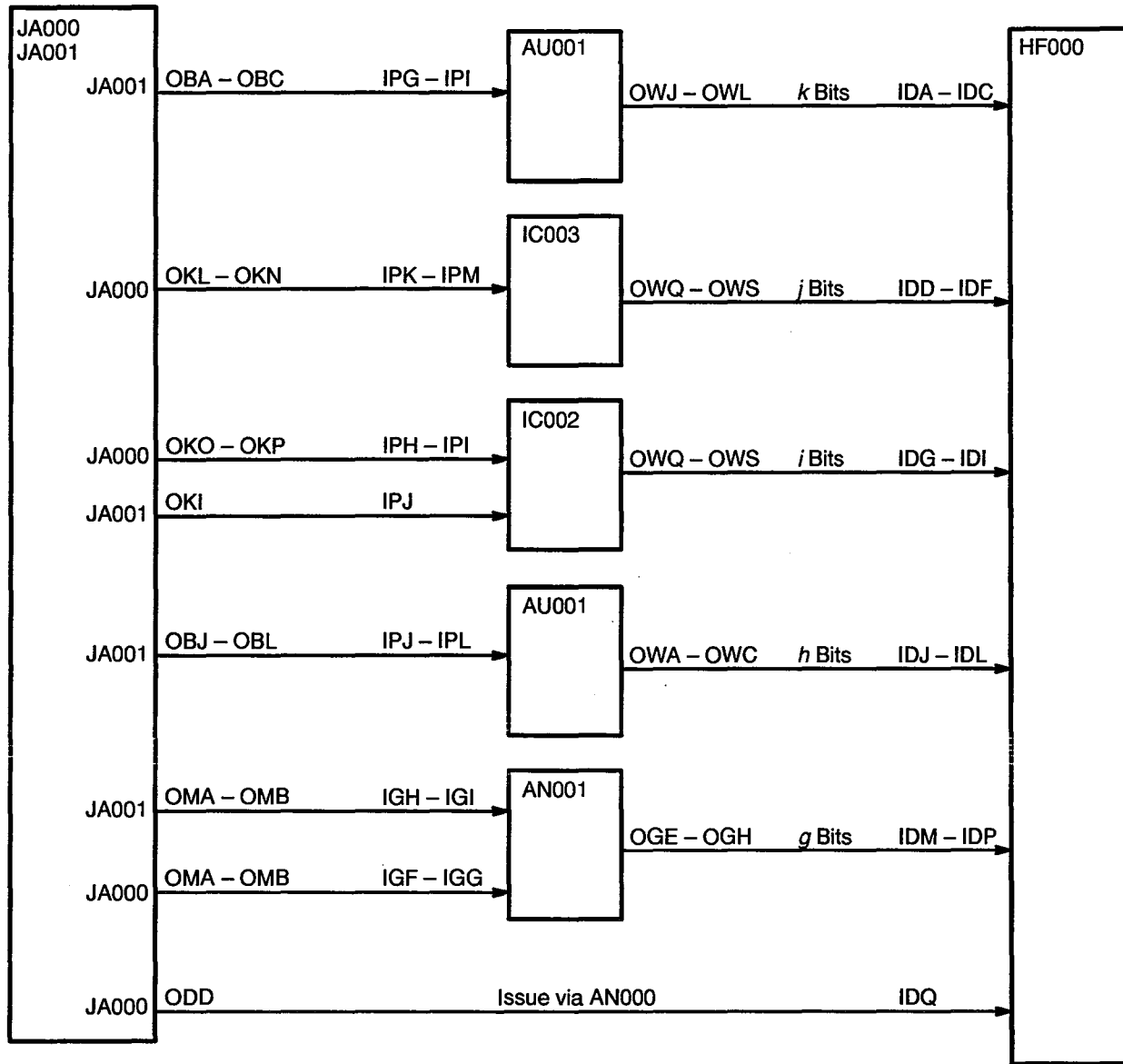


Figure 98. CIP Distribution to HF Option



# EXCHANGE

*No readstart in T90  
everything is done  
with interrupt*

The exchange mechanism in a CRAY T90 series computer system has the following features:

- Means of switching execution from program to program
- Exchange package – Block (40<sub>8</sub> words) of program parameters that:
  - Must be present in order for any program to execute; defines where and how the program runs
  - Must be 40<sub>8</sub> words long
  - Must reside in lower 2 MW of memory
  - Must start on a 40<sub>8</sub> word boundary

## Exchange Process

The exchange sequence is the process that deactivates the current exchange package and puts it into memory. It then loads a new exchange package from memory and activates it.

The CRAY T90 series systems have a new feature in the exchange package. This feature allows a process to exchange to either the address specified by the exchange address (XA) register or to one of five different addresses specified by one of the five exit address (EA) registers. With this capability, a user job could exchange to another user job, or could exchange to specific areas in the kernel, without first exchanging to the monitor.

Other features that are now implemented in the CRAY T90 series system include the following: when an exchange occurs, the CPU that exchanges out will retain the cluster number it was initially assigned unless the system is operating in C90 mode or unless AutoBCD (automatic broadcast cluster detach) is active. In addition, when a CPU is master cleared and then exchanged out, the pending interrupt bits are retained. This is done so that the maximum amount of information about the process is available. A second exchange sequence can retrieve this information.

If an exchange occurs and the program is in monitor mode, the monitor needs to save the B registers, T registers, shared registers, scalar (S) registers, and vector (V) registers. If the vector not used (VNU) bit is a 1, the V registers do not need to be saved. If the exchange is to another user job, it is up to the user to save the register values.

Four conditions cause an exchange sequence:

- Deadstart sequence (SIPI)
- Interrupt flag set (F register)
- Program exit (004000, 000000 instruction)
- Hardware error causing a flag to set, which causes an exchange

## Deadstart

A CRAY T90 series system does not use a deadstart signal or command; instead, the system uses **Set Interprocessor Interrupt (SIPI)** signals, via a 0014j1 instruction [send inter-CPU interrupt to CPU (Aj)] or, on an initial deadstart, a CPU loop controller function of 76g issued by the maintenance channel will start an exchange.

The sequence of events to start execution of MME:

- Set CPU MC
- Load data to memory address 0 via the maintenance channel.
- Issue a loop controller function of 176g via the maintenance channel to allow CPU maintenance instructions.
- Issue a loop controller function of 141g via the maintenance channel to allow cpu instruction exchange and halt.

The exchange package at location 0 goes into the CPU, and what was in the CPU goes to location 0. There is no fetch after this exchange.

- Drop CPU Master Clear via the maintenance channel.

- Issue the loop controller function of 76g via the maintenance channel.

The dropping of CPU Master Clear works as an enable; the function 76g must be present along with the Master Clear signal for the exchange to occur.

- Interrupted CPU exchanges to address 0, a fetch is done and issue starts.

In this case, because I/O is handled by the maintenance channel, the return path for output depends on how the sanity tree was configured. From this point, the initially started CPU could issue SIPI commands to the other CPUs.

## Interrupt Flag Set

In the CRAY T90 series system, each interrupt flag has an enable interrupt mode bit. The interrupt modes are enabled by the enabled interrupt mode (EIM) flag; an exchange to non-monitor mode sets the EIM flag.

An exchange to monitor mode clears the EIM flag. While the program is in monitor mode, a 001302 instruction sets the EIM flag, and an 001303 instruction clears the EIM flag.

Each CPU has an EIM flag. In monitor mode, the EIM flag is cleared and all interrupt modes are disabled, except enable flag on normal exit (FNX), enable flag on error exit (FEX), and enable interrupt on program range error (IPR); this provides a stable environment within monitor mode immediately following an exchange.

## Program Exit

Program exit occurs following the decode of instructions 000000 and 004000. Instruction 000000 is an error exit instruction, and instruction 004000 is a normal exit.

## Exchange Sequence

---

Before a CPU can perform an exchange, the CPU must first finish all active instructions. If a test and set instruction (0034jk) is in the next instruction parcel (NIP) or entering the current instruction parcel (CIP), the program (P) register is decremented by 2, or by 1 if the test and set instruction is in the CIP or NIP respectively. The JA option transmits a

signal to the BT options that decrements the P register before it is loaded into memory. The JA then waits until the condition is resolved to advance P. Memory must also be quiet, and all memory writes must have completed.

The processor performing the exchange clears out the buffer valid bits and buffer counter. Clearing the buffer valid bits causes a fetch to occur after the exchange has completed. Clearing the instruction buffer address register (IBAR) counter causes the data that was fetched from memory to be loaded into instruction buffer 0 first. Also, issuing a 0051jk instruction clears the buffer valid bits. The 0051jk is a maintenance instruction that loads the P register from Bjk and invalidates the instruction buffers if the CPU is in maintenance mode (MM).

## Exchange Package Descriptions

---

Refer to Figure 99 for an illustration of the exchange package. The exchange parameters are located on two options: HD000 and HD001. HD000 handles bits 0 through 31 for words 0 through 17, and HD001 handles bits 32 through 63 for words 0 through 17.

**P register** – program register, word 10 bits 0 through 31

The P register contains 32 bits, the lower 2 bits of which are used for parcel selects. The P register contains bits –2 through 29, which allow 1 gigaword of memory to be addressed.

**Modes** – MM, BDM, ESL, TRI, SCE, BDD word 11, bits 0 through 7

The modes tell the program what it can or cannot do, thereby determining what effect the instructions issued will have on the program.

**MM** – monitor mode, word 11, bit 0

Certain instructions are privileged to MM: controlling the channel, setting the real-time clock, setting the programmable clock, and so on. These instructions perform specialized functions that are useful to the operating system. If an MM instruction issues while the CPU is not in MM, it is treated as a no-operation instruction. If an MM instruction issues while the IMI flag is set, the MII flag sets, causing an exchange.

**BDM – bidirectional memory, word 11, bit 1**

When BDM is set, block reads and writes may occur concurrently.

**ESL – enable second vector logical, word 11, bit 2**

If ESL is set and any 140ijk through 145ijk instructions issue, the instruction is routed to the second vector logical unit. If ESL = 0, the second vector logical unit is not used. The second vector logical unit is used before the full vector logical unit if a choice exists.

**TRI – Triton mode, word 11, bit 3**

The Triton mode allows the new instruction to run in the CRAY T90 series system. If the Triton mode bit equals a 0, then the instruction will run only CRAY C90 instructions.

**SCE – scalar cache enabled, word 11, bit 2<sup>4</sup>**

If SCE is set to a 1, onboard scalar cache is enabled.

**BDD – bidirectional memory disable, word 11, bit 2<sup>7</sup>**

When BDD is set to a 1, bidirectional block reads and writes are disabled.

**Status (VNU, FPS, WS, PS), word 12, bit 0 through 3**

The status register reflects the condition of the CPU at the time of an exchange. The bits in the status field are set during program execution and are not user selectable.

**VNU – vectors not used, word 12, bit 3**

After a program has been exchanged into memory, the B and T registers must be saved as well as the SB, ST, and SM registers of the cluster that the program is using. If the VNU bit is equal to 1, then this indicates that the vector registers were not used so the vector registers do not need to be saved. However, if the VNU bit is 0, then the vector registers must be saved as well. The VNU bit is set when a 077xxx or a 140 through 177xxx instruction issues.

	63 0	48 47 15, 16	32 31 31, 32	16 15 47, 48	0 63
0	LAT 0 Modes RW X C	LAT 0 Logical Limit		LAT 0 Logical Base	
1	LAT 1 Modes RW X C	LAT 1 Logical Limit		LAT 1 Logical Base	
2	LAT 2 Modes RW X C	LAT 2 Logical Limit		LAT 2 Logical Base	
3	LAT 3 Modes RW X C	LAT 3 Logical Limit		LAT 3 Logical Base	
4	LAT 4 Modes RW X C	LAT 4 Logical Limit		LAT 4 Logical Base	
5	LAT 5 Modes RW X C	LAT 5 Logical Limit		LAT 5 Logical Base	
6	LAT 6 Modes RW X C	LAT 6 Logical Limit		LAT 6 Logical Base	
7	LAT 7 Modes RW X C	LAT 7 Logical Limit		LAT 7 Logical Base	
10	LAT 0 Modes RW X D	LAT 0 Physical Bias		P Register	
11	LAT 1 Modes RW X D	LAT 1 Physical Bias		Interrupt Modes R U F O P E B C M R I I P D M N A P M P R R X P M C T P O C L I X M Modes B S T E B M D -- C R S D M D E I L M	
12	LAT 2 Modes RW X D	LAT 2 Physical Bias		Interrupt Flags R M F O P E B M M R I I P D M N A P E P R R E P E C T C O C L I E M E U E E X I C U I P I I I X I Status V F W B N P S M U S L	
13	LAT 3 Modes RW X D	LAT 3 Physical Bias		Cluster Number: 7 0 Processor Number: 6 0 Vector Length: 7 0	
14	LAT 4 Modes RW X D	LAT 4 Physical Bias			
15	LAT 5 Modes RW X D	LAT 5 Physical Bias		Exit Address 3: 20 5 20 Exit Address 4: 20 5	
16	LAT 6 Modes RW X D	LAT 6 Physical Bias		Exit Address 1: 20 5 20 Exit Address 2: 20 5	
17	LAT 7 Modes RW X D	LAT 7 Physical Bias		Exchange Address: 20 5 20 Exit Address 0: 20 5	

Words 20 - 27: A Registers 0 - 7  
 Words 30 - 37: S Registers 0 - 7

Figure 99. Exchange Package

**FPS – floating-point status, word 12, bit 2**

A floating-point error sets the FPS flag regardless of the state of the floating-point error flag (FPE). The FPE flag sets when an underflow or overflow condition exists in the floating-point functional units.

The FPS bit is cleared whenever the interrupt on floating-point error (IFP) mode bit is set or cleared by a 002100 or 002200 instruction.

The FPS bit is also cleared when the bit matrix loaded (BML) flag is cleared; the BML flag is cleared when a 002210 instruction issues.

**WS – waiting on semaphore, word 12, bit 1**

The WS bit sets when a 0034jk instruction is in CIP and holding issue.

**BML – bit matrix loaded, word 12, bit 0**

The BML bit indicates the B<sup>t</sup> (B transposed) registers have been successfully loaded by a 1740j4 instruction.

**Interrupt modes, word 11, bits 15 through 31**

Refer to Table 30 for a list of the bit assignments for the modes field in the exchange package. All modes except IPR, FEX, and FNX must be enabled by the EIM flag to be effective. The EIM flag sets on an exchange to nonmonitor mode and clears on an exchange to monitor mode. The EIM flag enables interrupt modes if set.

The EIM bit can be set or cleared by a 001302 or a 001303 instruction, respectively.

Table 30. Interrupt Modes Register Bit Assignments

Word	Binary Exponent	Acronym	Name
11	31	IRP	Interrupt on Register Parity Error
11	30	IUM	Interrupt on Uncorrectable Memory Error
11	29	IFP	Interrupt on Floating-point Error
11	28	IOR	Interrupt on Operand Range Error
11	27	IPR	Interrupt on Program Range Error
11	26	FEX	Enable Flag on Error Exit (does not disable exchange)
11	25	IBP	Interrupt on Breakpoint
11	24	ICM	Interrupt on Correctable Memory Error
11	23	IMC	Interrupt on MCU Interrupt
11	22	IRT	Interrupt on Real-time Interrupt
11	21	IIP	Interrupt on Interprocessor Interrupt
11	20	IIO	Interrupt on I/O
11	19	IPC	Interrupt on Programmable Clock
11	18	IDL	Interrupt on Deadlock
11	17	IMI	Interrupt on 001 $ij \neq 0$ or 033 instruction
11	16	FNX	Enable Flag on Normal Exit (does not disable exchange)
11	15	IAM	Interrupt on Address Multiply Range Error



Refer to Table 31 for a list of the bit assignments for the interrupt flags field in the exchange package.

Table 31. Flag Register Bit Assignments

Word	Binary Exponent	Acronym	Name
12	31	RPE	Register Parity Error
12	30	MEU	Uncorrectable Memory Error
12	29	FPE	Floating-point Error
12	28	ORE	Operand Range Error
12	27	PRE	Program Range Error
12	26	EEX	Error Exit (000 issued)
12	25	BPI	Breakpoint Interrupt
12	24	MEC	Correctable Memory Error
12	23	MCU	MCU Interrupt
12	22	RTI	Real-time Interrupt
12	21	ICP	Interrupt from Internal CPU
12	20	IOI	I/O Interrupt (if IIO and SIE) <sup>†</sup>
12	19	PCI	Programmable Clock Interrupt
12	18	DL	Deadlock Interrupt
12	17	MII	001 $ij \neq 0$ or 033 Instruction Interrupt (if IMI and not MM)
12	16	NEX	Normal Exit (004 issued)
12	15	AMI	Address Multiply Interrupt

<sup>†</sup> SIE = System I/O interrupt enabled.

**VL** – vector length, word 13, bits 0 through 7

The VL register holds the content of the VL register. The 8-bit field contains the number of elements to be operated on in the vector register. In a CRAY T90 series system, if VL = 000 or VL = 200, all 200<sub>8</sub> vector elements are used within the vector register.

**XA** – exchange address, word 17, bits 16 through 31

The 16-bit field specifies the address of the first word of the next exchange package. This exchange package is loaded when any one of the following conditions occurs:

- An interrupt occurs that sets any of the following flags: RPE, MEU, FPE, OPR, BPI, MEC, MCU, RTI, ICP, IOI, PCI, DL, MII, NEX, or AMI
- A 000 is issued
- A 0040*jk* is issued with *k* being an illegal value (5, 6, or 7)

The XA field contains only bits 5 through 20. The lower bits are assumed to be 0's.

**EXIT Address 0 through 4**, words 15, 16, 17 bits 0 through 31

Each of the five 16-bit fields specifies the starting address of a 32-word exchange package. The *k* field of the 0040*jk* instruction specifies the exchange package to use. Only *k* fields equal to 0 through 4 are valid; if an invalid value is used, the exchange is to the XA address. Exit Address (EA) 0 is expected to be used for normal exits to maintain compatibility with existing systems.

Each EA field contains only bits 5 through 20. The lower bits are assumed to be 0's.

**CLN** – cluster number, word 13, bits 24 through 31

The CLN contains a 8-bit field. There are up to 36<sub>8</sub> clusters in the system, depending on the system configuration.

PPN – Processor number, word 13, bits 16 through 22

The contents of the 7-bit field in the exchange packages show the logical number of the CPU in which the exchange was executed. The maximum number is 127.

LATS – Words 0 through 17. Refer to the exchange package diagram for bit layouts.

Each LAT has four associated fields; Table 32 identifies those fields.

Table 32. LAT Fields

Field Name	Description
Logical Base	First logical address of this LAT
Logical Limit	Last address +1 of this LAT
Physical Bias	Physical bias = Physical base address – Logical base address
Modes	The controlling bits for each LAT R(ead), W(rite), X(ecute), C(achable), D(irty)

The use of LATs allows programs to share memory space. For example, two user jobs could reference the same library routine in memory while keeping their local code private.



# REAL-TIME CLOCK PROGRAMMABLE CLOCK INTERRUPT STATUS REGISTER PERFORMANCE MONITOR

Refer to the following subsections for information about the real-time clock, programmable clock interrupt, status register, and the performance monitor.

## Real-time Clock

---

A CRAY T90 series computer system contains one 64-bit real-time clock (RTC) per central processing unit (CPU). The RTC is synchronized when a CPU issues a 0014j0 instruction. The 0014j0 instruction causes all CPUs in the same cluster to be loaded with the contents of Sj. The RTC is located on two HD options, each of which handles 32 bits. The HD000 option handles bits 0 through 31; the HD001 option handles bits 32 through 63.

HD000 will detect a carry, out of the RTC, at a count of 3777777776 during normal operation. HD001 then increments the upper bits during the next clock period, and HD000 suppresses any toggles.

The RTC is incremented once every clock period. The RTC allows for clock-period timing of program execution. When the machine is deadstarted, the RTC must be loaded in order to synchronize all the CPUs. If they are not synchronized, each CPU will have a different RTC value.

Writing to the RTC with the 0014j0 instruction sends a copy of the Sj register from the CPU issuing the instruction to all RTC registers via the issue paths of the shared registers. Reading the RTC with a 072i00 instruction copies the RTC register of the CPU that issued the 072i00 instruction into the scalar registers.

Refer to Figure 100 for an RTC and programmable clock interrupt (PCI) block diagram.

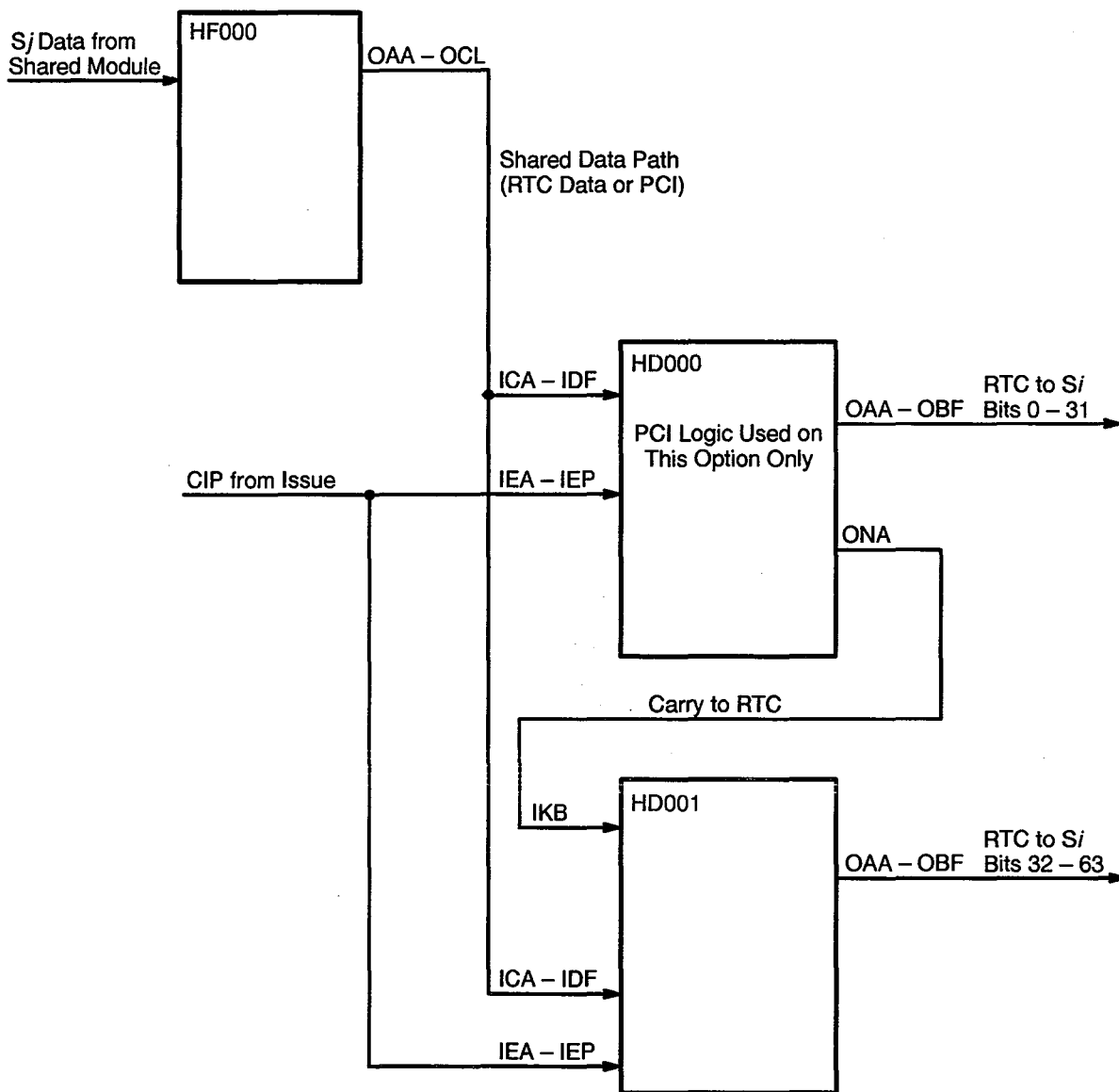


Figure 100. RTC and PCI Block Diagram

## Programmable Clock

Each CPU has one programmable clock (PC), which is a 32-bit counter. The programmable clock decrements every clock period; the clock is located on the HD000 option.

The programmable clock is loaded by the 0014j4 instruction when the program is in monitor mode. When the programmable clock equals zero, an interrupt request (PCI) is generated. To generate a PCI, the IPC mode bit must be set. In user mode, IPC must have been set in the user's exchange package. If the CPU is in monitor mode, either IPC was set in

the monitor's exchange package, or a 001406 instruction was issued. The interrupt request remains set until cleared by a 001405 instruction. If the CPU is in monitor mode, and if the interrupt request is not desired, use a 001407 instruction to disable the IPC mode bit.

The PCI request is enabled and disabled on the HD option, which contains the exchange parameters.

## RTC and PC Instructions

---

Refer to Table 33 for a list of the RTC and PC instructions.

Table 33. RTC and PC Instructions

Instruction	CAL	Description
0014j0 †	RT Sj	Enter RTC register with Sj
072j00	Si RT	Transmit RTC to Si
0014j4 †	PCI Sj	Transmit Sj to programmable clock
001405 †	CCI	Clear PCI request
001406 †	ECl	Enable PCI request
001407 †	DCI	Disable PCI request

† Data cache is a monitor mode instruction.

## Performance Monitor

---

The performance monitor (PM) is normally used to monitor software performance. With the results of the performance monitor, a programmer can determine how efficiently a program is running in the system. If, for example, the program is performing too many instruction fetches or too many hold issue conditions are occurring, the programmer can review the program structure and modify it to minimize these occurrences.

Each CPU contains a performance monitor; because each CPU is identical, all references in this section pertain to a single CPU. Each CPU contains 32 performance counters and each counter is 48 bits wide. Table 34 shows which event each counter monitors. Each counter increments each time a particular event occurs in the CPU while the CPU is in nonmonitor mode (IMI bit is not set). The counters related to memory references may be incremented by as many as eight times per clock period (CP). Counters related to vector operations are incremented by the value in the vector length register at the time the instruction issues.

Table 34. Performance Monitor

Counter	Event Monitored	Instructions	Increments
	Number of:		
0	Clock periods monitored		+1
1	Instructions issued		+1
2	Clock periods holding issue		+1
3	Instruction fetches		+1
4	CPU memory references (ports A, B, C)		+8
5	Clock periods for references (ports A, B,C)		+2047
6	I/O memory references (port D, I/O only)		+2
7	Cache misses		+1
	Holding issue on:		
10	A registers and access conflicts		+1
11	S registers and access conflicts		+1
12	V registers		+1
13	B/T registers		+1
14	Functional units		+1
15	Shared registers		+1
16	Memory ports		+1
17	Number of cache hits		+1
	Number of instructions:		
20	Instructions 000000 through 004000	000 – 004	+1
21	Branches	005 – 017	+1
22	Address instructions	02x, 030 – 033, EIS 042 – 057 ,073i20, 073i30	+1
23	B/T memory instructions	034 – 037	+1
24	Scalar instructions	040 – 043, 071 – 077 except 073i20, 073i30	+1
25	Scalar integer instructions	044 – 061, 070ij6	+1
26	Scalar floating-point instructions	062 – 070	+1
27	S/A memory instructions	10x – 13x	+1
	Number of operations:		
30	Vector logical	070ij1, 140 – 147, 1740j4 – 1740j6, 175	+VL
31	Vector shifts, pop., leading zero	150 – 153, 174xx (1 – 3)	+VL
32	Vector integer adds	154 – 157	+VL
33	Vector floating-point multiplies	160 – 167	+VL
34	Vector floating-point adds	170 – 173	+VL
35	Vector floating-point reciprocals	174xx0	+VL
36	Vector memory reads	176	+VL
37	Vector memory writes	177	+VL



## Performance Monitor Instructions

---

Table 35 lists all the instructions associated with the performance monitor.

Table 35. Performance Monitor Instructions

Instruction	CAL	Description
001500		Clear all performance counters
073 <i>ij</i> 1	<i>Si</i> <i>SRj</i>	Transmit ( <i>SRj</i> ) to <i>Si</i> (monitor mode only for $j = 2 - 7$ )
073 <i>i</i> 05	SR0 <i>Si</i>	Transmit ( <i>Si</i> ) bits 48 – 52 to SR0
073 <i>i</i> 25	SR2 <i>Si</i>	Advance performance monitor pointer
073 <i>i</i> 75	SR7 <i>Si</i>	Transmit ( <i>Si</i> ) to maintenance channel

## Clearing the Performance Counters

---

Instruction 001500 clears all performance counters. This instruction must be issued while the CPU is in monitor mode in order for the instruction to operate correctly.

## Reading the Performance Monitor

---

The performance monitor is read with the 073*i*21 and 073*i*31 instructions. Each counter is read 48 bits at a time and requires that two instructions be issued to read all the counters. The 48 bits of the counter read are stored in the *Si* register. When the 073*i*21 instruction is issued, counters 0 through 17 are sent to *Si*. The 073*i*31 instruction, when issued, reads counters 20 through 37 and sends the bits to *Si*.

The system hardware requires a minimum of 3 CPs between issuing 073*ix*1 instructions. Also, the PM Busy Status (PMBY) bit (bit 47 of SR0) must be cleared before reading the counters. If the 3-CP wait is not written into the program, an undeterminable corruption of performance monitor data occurs.

## Performance Monitor Block Diagram

---

Refer to Figure 101 for the performance monitor block diagram. The performance monitor is composed of the HF000, HD000, and HD001 options. The HF000 option contains the lower bits (0 through 31) and the HD000 and HD001 options contain the upper bits (32 through 47) for all 32 counters; there is one counter for each event tracked by the performance monitor. These 48-bit counters are incremented as each event occurs, as long as the CPU is not in monitor mode.

## Status Register

---

A CRAY T90 series computer system has eight status registers, which are located on the HD and HF options. The status register is no longer part of the exchange package as it was in previous systems. Figure 102 shows the status register format and bit assignments of each register. The status registers are read by the 073ij1 instruction.

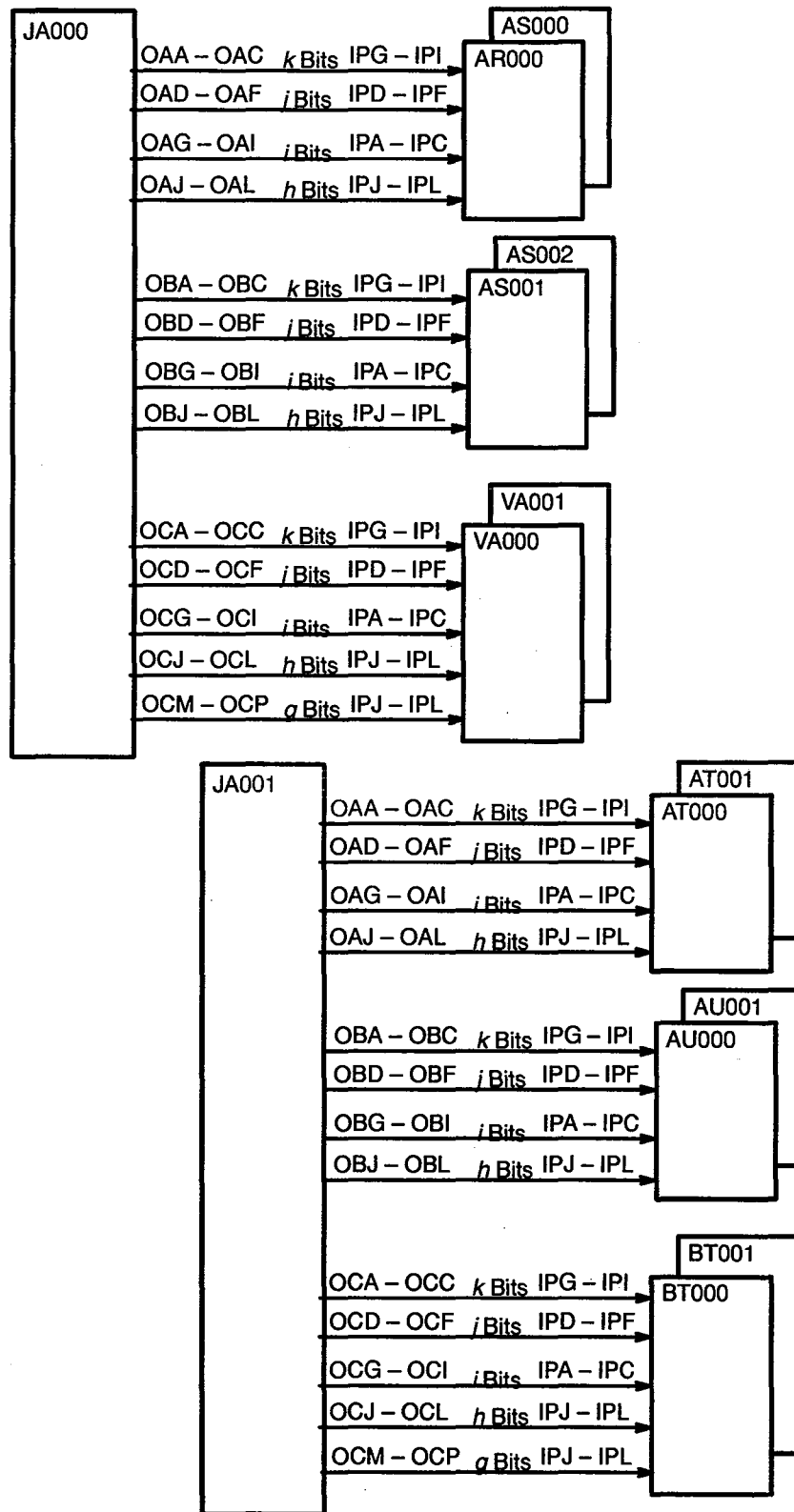


Figure 93. Instruction Data Distribution A/S/B/T Registers

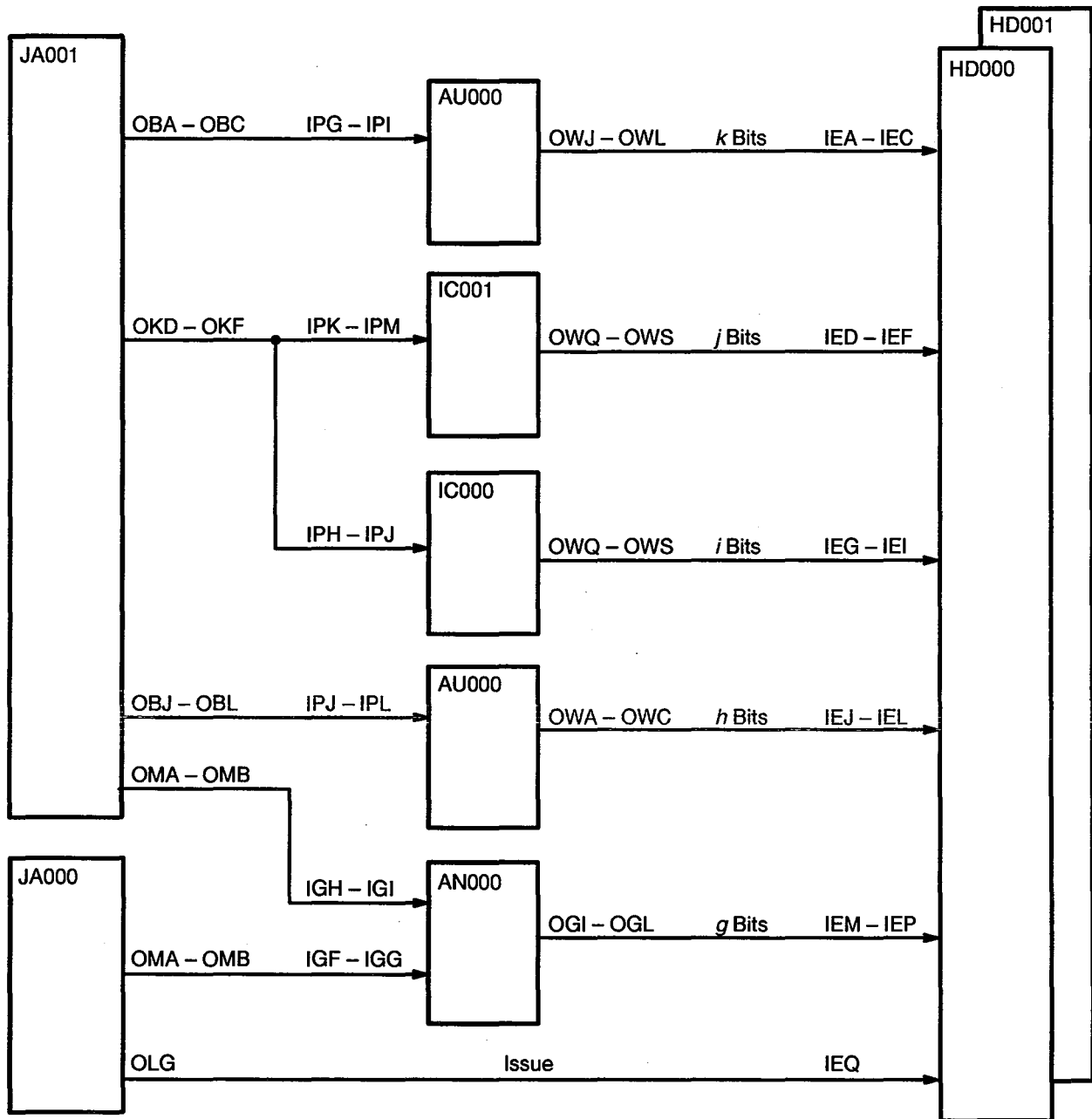


Figure 94. CIP Distribution

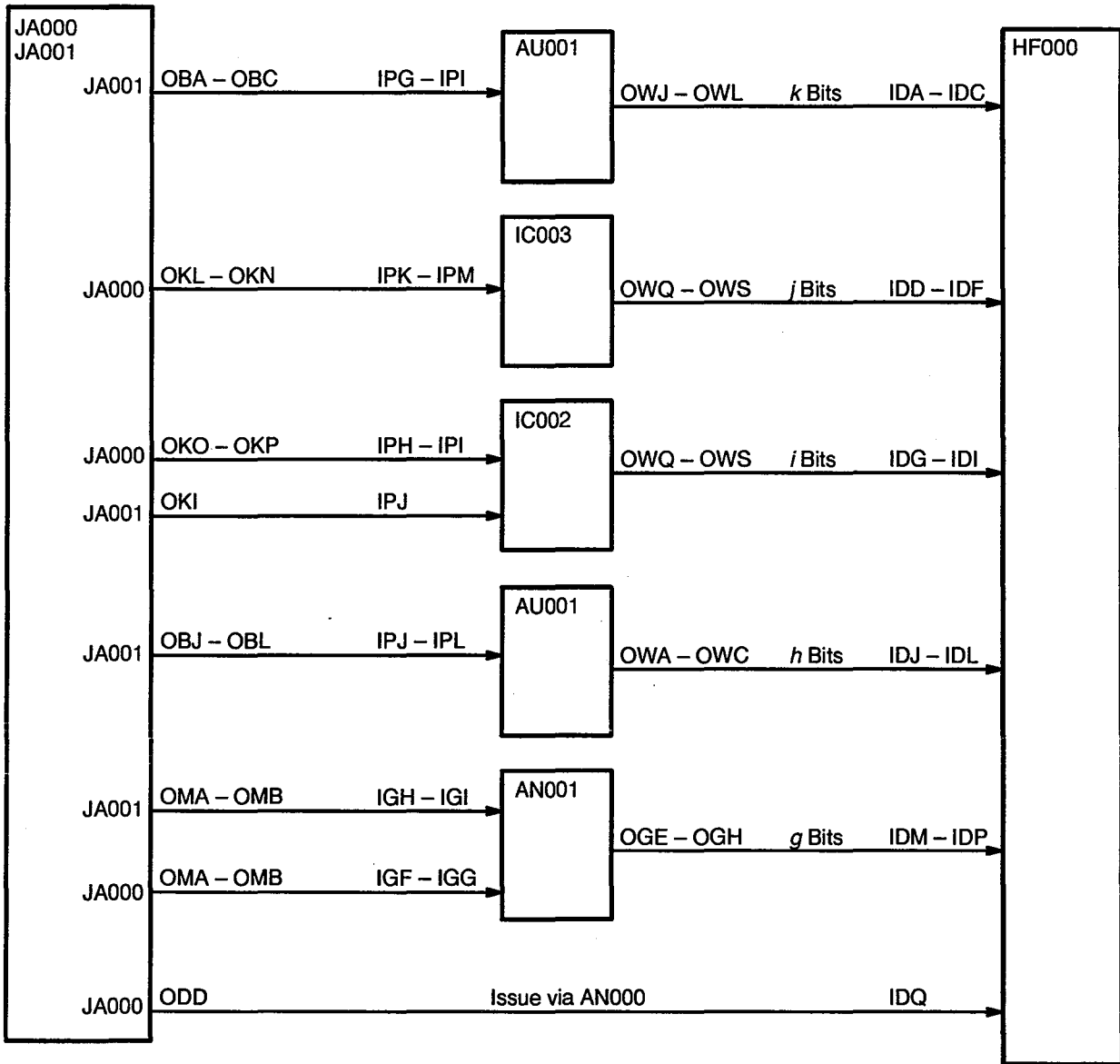


Figure 95. CIP Distribution to HF Option

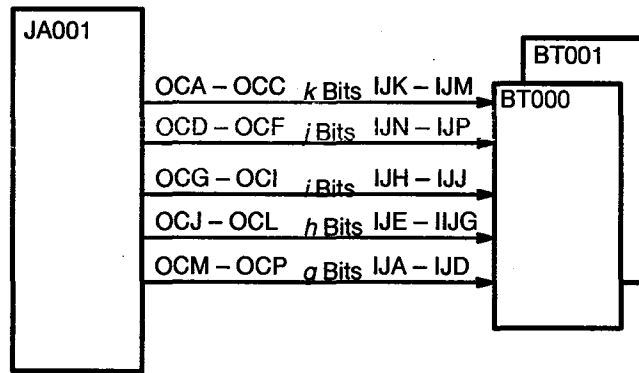
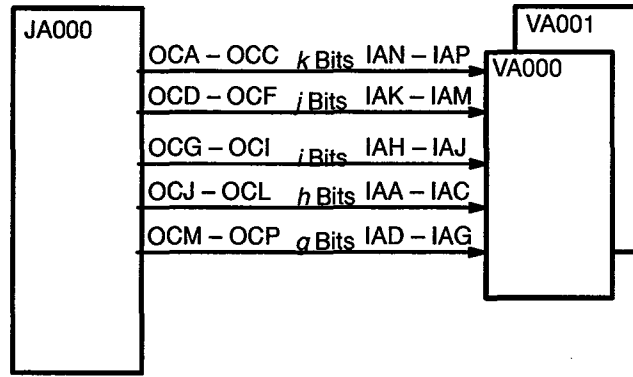


Figure 96. Instruction Data Distribution to VA and BT Options

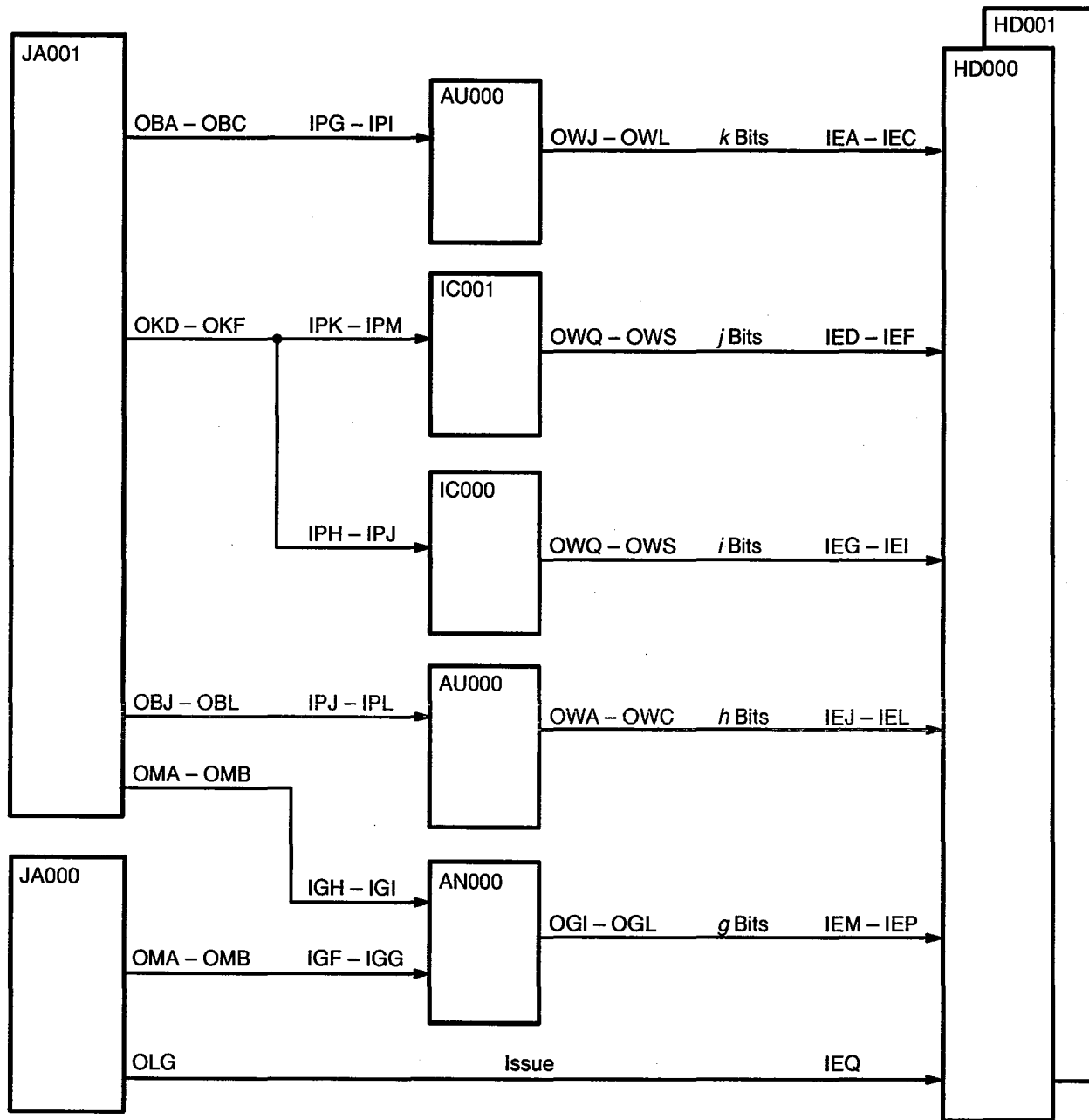


Figure 97. CIP Distribution

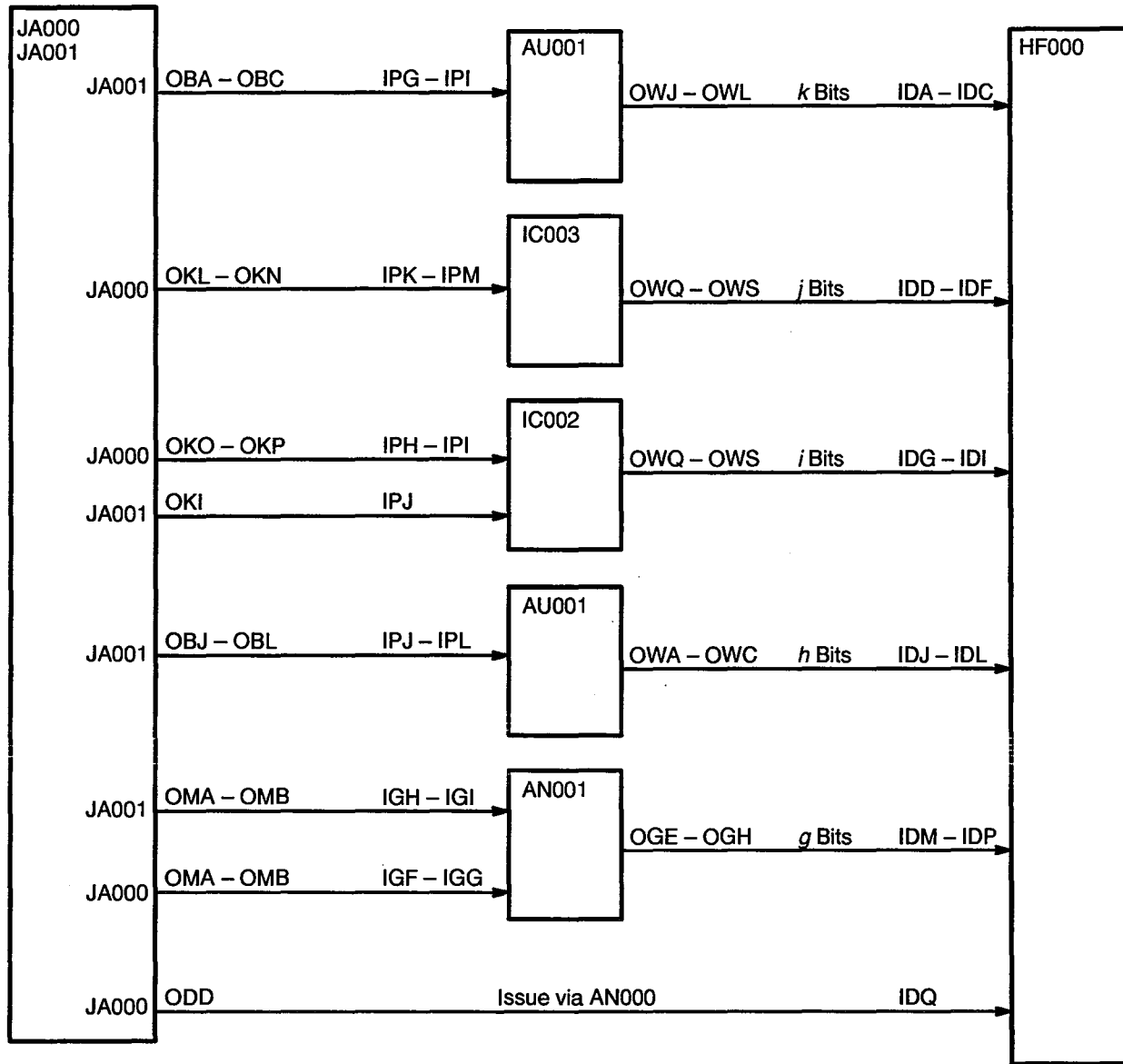


Figure 98. CIP Distribution to HF Option



# EXCHANGE

*No readstart in T90  
everything is done  
with interrupt*

The exchange mechanism in a CRAY T90 series computer system has the following features:

- Means of switching execution from program to program
- Exchange package – Block (40<sub>8</sub> words) of program parameters that:
  - Must be present in order for any program to execute; defines where and how the program runs
  - Must be 40<sub>8</sub> words long
  - Must reside in lower 2 MW of memory
  - Must start on a 40<sub>8</sub> word boundary

## Exchange Process

The exchange sequence is the process that deactivates the current exchange package and puts it into memory. It then loads a new exchange package from memory and activates it.

The CRAY T90 series systems have a new feature in the exchange package. This feature allows a process to exchange to either the address specified by the exchange address (XA) register or to one of five different addresses specified by one of the five exit address (EA) registers. With this capability, a user job could exchange to another user job, or could exchange to specific areas in the kernel, without first exchanging to the monitor.

Other features that are now implemented in the CRAY T90 series system include the following: when an exchange occurs, the CPU that exchanges out will retain the cluster number it was initially assigned unless the system is operating in C90 mode or unless AutoBCD (automatic broadcast cluster detach) is active. In addition, when a CPU is master cleared and then exchanged out, the pending interrupt bits are retained. This is done so that the maximum amount of information about the process is available. A second exchange sequence can retrieve this information.

If an exchange occurs and the program is in monitor mode, the monitor needs to save the B registers, T registers, shared registers, scalar (S) registers, and vector (V) registers. If the vector not used (VNU) bit is a 1, the V registers do not need to be saved. If the exchange is to another user job, it is up to the user to save the register values.

Four conditions cause an exchange sequence:

- Deadstart sequence (SIPI)
- Interrupt flag set (F register)
- Program exit (004000, 000000 instruction)
- Hardware error causing a flag to set, which causes an exchange

## Deadstart

A CRAY T90 series system does not use a deadstart signal or command; instead, the system uses **Set Interprocessor Interrupt (SIPI)** signals, via a 0014j1 instruction [send inter-CPU interrupt to CPU (Aj)] or, on an initial deadstart, a CPU loop controller function of 76g issued by the maintenance channel will start an exchange.

The sequence of events to start execution of MME:

- Set CPU MC
- Load data to memory address 0 via the maintenance channel.
- Issue a loop controller function of 176g via the maintenance channel to allow CPU maintenance instructions.
- Issue a loop controller function of 141g via the maintenance channel to allow cpu instruction exchange and halt.

The exchange package at location 0 goes into the CPU, and what was in the CPU goes to location 0. There is no fetch after this exchange.

- Drop CPU Master Clear via the maintenance channel.

- Issue the loop controller function of 76g via the maintenance channel.

The dropping of CPU Master Clear works as an enable; the function 76g must be present along with the Master Clear signal for the exchange to occur.

- Interrupted CPU exchanges to address 0, a fetch is done and issue starts.

In this case, because I/O is handled by the maintenance channel, the return path for output depends on how the sanity tree was configured. From this point, the initially started CPU could issue SIPI commands to the other CPUs.

## Interrupt Flag Set

In the CRAY T90 series system, each interrupt flag has an enable interrupt mode bit. The interrupt modes are enabled by the enabled interrupt mode (EIM) flag; an exchange to non-monitor mode sets the EIM flag.

An exchange to monitor mode clears the EIM flag. While the program is in monitor mode, a 001302 instruction sets the EIM flag, and an 001303 instruction clears the EIM flag.

Each CPU has an EIM flag. In monitor mode, the EIM flag is cleared and all interrupt modes are disabled, except enable flag on normal exit (FNX), enable flag on error exit (FEX), and enable interrupt on program range error (IPR); this provides a stable environment within monitor mode immediately following an exchange.

## Program Exit

Program exit occurs following the decode of instructions 000000 and 004000. Instruction 000000 is an error exit instruction, and instruction 004000 is a normal exit.

## Exchange Sequence

---

Before a CPU can perform an exchange, the CPU must first finish all active instructions. If a test and set instruction (0034jk) is in the next instruction parcel (NIP) or entering the current instruction parcel (CIP), the program (P) register is decremented by 2, or by 1 if the test and set instruction is in the CIP or NIP respectively. The JA option transmits a

signal to the BT options that decrements the P register before it is loaded into memory. The JA then waits until the condition is resolved to advance P. Memory must also be quiet, and all memory writes must have completed.

The processor performing the exchange clears out the buffer valid bits and buffer counter. Clearing the buffer valid bits causes a fetch to occur after the exchange has completed. Clearing the instruction buffer address register (IBAR) counter causes the data that was fetched from memory to be loaded into instruction buffer 0 first. Also, issuing a 0051jk instruction clears the buffer valid bits. The 0051jk is a maintenance instruction that loads the P register from Bjk and invalidates the instruction buffers if the CPU is in maintenance mode (MM).

## Exchange Package Descriptions

---

Refer to Figure 99 for an illustration of the exchange package. The exchange parameters are located on two options: HD000 and HD001. HD000 handles bits 0 through 31 for words 0 through 17, and HD001 handles bits 32 through 63 for words 0 through 17.

**P register – program register, word 10 bits 0 through 31**

The P register contains 32 bits, the lower 2 bits of which are used for parcel selects. The P register contains bits –2 through 29, which allow 1 gigaword of memory to be addressed.

**Modes – MM, BDM, ESL, TRI, SCE, BDD word 11, bits 0 through 7**

The modes tell the program what it can or cannot do, thereby determining what effect the instructions issued will have on the program.

**MM – monitor mode, word 11, bit 0**

Certain instructions are privileged to MM: controlling the channel, setting the real-time clock, setting the programmable clock, and so on. These instructions perform specialized functions that are useful to the operating system. If an MM instruction issues while the CPU is not in MM, it is treated as a no-operation instruction. If an MM instruction issues while the IMI flag is set, the MII flag sets, causing an exchange.

**BDM** – bidirectional memory, word 11, bit 1

When BDM is set, block reads and writes may occur concurrently.

**ESL** – enable second vector logical, word 11, bit 2

If ESL is set and any 140*ijk* through 145*ijk* instructions issue, the instruction is routed to the second vector logical unit. If ESL = 0, the second vector logical unit is not used. The second vector logical unit is used before the full vector logical unit if a choice exists.

**TRI** – Triton mode, word 11, bit 3

The Triton mode allows the new instruction to run in the CRAY T90 series system. If the Triton mode bit equals a 0, then the instruction will run only CRAY C90 instructions.

**SCE** – scalar cache enabled, word 11, bit 2<sup>4</sup>

If SCE is set to a 1, onboard scalar cache is enabled.

**BDD** – bidirectional memory disable, word 11, bit 2<sup>7</sup>

When BDD is set to a 1, bidirectional block reads and writes are disabled.

**Status (VNU, FPS, WS, PS)**, word 12, bit 0 through 3

The status register reflects the condition of the CPU at the time of an exchange. The bits in the status field are set during program execution and are not user selectable.

**VNU** – vectors not used, word 12, bit 3

After a program has been exchanged into memory, the B and T registers must be saved as well as the SB, ST, and SM registers of the cluster that the program is using. If the VNU bit is equal to 1, then this indicates that the vector registers were not used so the vector registers do not need to be saved. However, if the VNU bit is 0, then the vector registers must be saved as well. The VNU bit is set when a 077xxx or a 140 through 177xxx instruction issues.

	63 0		48 47 15, 16		32 31 31, 32		16 15 47, 48		0 63
0	LAT 0 Modes RW X C		LAT 0 Logical Limit				LAT 0 Logical Base		
1	LAT 1 Modes RW X C		LAT 1 Logical Limit				LAT 1 Logical Base		
2	LAT 2 Modes RW X C		LAT 2 Logical Limit				LAT 2 Logical Base		
3	LAT 3 Modes RW X C		LAT 3 Logical Limit				LAT 3 Logical Base		
4	LAT 4 Modes RW X C		LAT 4 Logical Limit				LAT 4 Logical Base		
5	LAT 5 Modes RW X C		LAT 5 Logical Limit				LAT 5 Logical Base		
6	LAT 6 Modes RW X C		LAT 6 Logical Limit				LAT 6 Logical Base		
7	LAT 7 Modes RW X C		LAT 7 Logical Limit				LAT 7 Logical Base		
10	LAT 0 Modes RW X D		LAT 0 Physical Bias				P Register		
11	LAT 1 Modes RW X D		LAT 1 Physical Bias				Interrupt Modes		Modes
12	LAT 2 Modes RW X D		LAT 2 Physical Bias				Interrupt Flags		Status
13	LAT 3 Modes RW X D		LAT 3 Physical Bias				Cluster Number	Processor Number	Vector Length
14	LAT 4 Modes RW X D		LAT 4 Physical Bias						
15	LAT 5 Modes RW X D		LAT 5 Physical Bias				Exit Address 3	Exit Address 4	
16	LAT 6 Modes RW X D		LAT 6 Physical Bias				Exit Address 1	Exit Address 2	
17	LAT 7 Modes RW X D		LAT 7 Physical Bias				Exchange Address	Exit Address 0	

Words 20 - 27: A Registers 0 - 7  
 Words 30 - 37: S Registers 0 - 7

Figure 99. Exchange Package

**FPS – floating-point status, word 12, bit 2**

A floating-point error sets the FPS flag regardless of the state of the floating-point error flag (FPE). The FPE flag sets when an underflow or overflow condition exists in the floating-point functional units.

The FPS bit is cleared whenever the interrupt on floating-point error (IFP) mode bit is set or cleared by a 002100 or 002200 instruction.

The FPS bit is also cleared when the bit matrix loaded (BML) flag is cleared; the BML flag is cleared when a 002210 instruction issues.

**WS – waiting on semaphore, word 12, bit 1**

The WS bit sets when a 0034jk instruction is in CIP and holding issue.

**BML – bit matrix loaded, word 12, bit 0**

The BML bit indicates the B<sup>t</sup> (B transposed) registers have been successfully loaded by a 1740j4 instruction.

**Interrupt modes, word 11, bits 15 through 31**

Refer to Table 30 for a list of the bit assignments for the modes field in the exchange package. All modes except IPR, FEX, and FNX must be enabled by the EIM flag to be effective. The EIM flag sets on an exchange to nonmonitor mode and clears on an exchange to monitor mode. The EIM flag enables interrupt modes if set.

The EIM bit can be set or cleared by a 001302 or a 001303 instruction, respectively.

Table 30. Interrupt Modes Register Bit Assignments

Word	Binary Exponent	Acronym	Name
11	31	IRP	Interrupt on Register Parity Error
11	30	IUM	Interrupt on Uncorrectable Memory Error
11	29	IFP	Interrupt on Floating-point Error
11	28	IOR	Interrupt on Operand Range Error
11	27	IPR	Interrupt on Program Range Error
11	26	FEX	Enable Flag on Error Exit (does not disable exchange)
11	25	IBP	Interrupt on Breakpoint
11	24	ICM	Interrupt on Correctable Memory Error
11	23	IMC	Interrupt on MCU Interrupt
11	22	IRT	Interrupt on Real-time Interrupt
11	21	IIP	Interrupt on Interprocessor Interrupt
11	20	IIO	Interrupt on I/O
11	19	IPC	Interrupt on Programmable Clock
11	18	IDL	Interrupt on Deadlock
11	17	IMI	Interrupt on 001 $ij \neq 0$ or 033 instruction
11	16	FNX	Enable Flag on Normal Exit (does not disable exchange)
11	15	IAM	Interrupt on Address Multiply Range Error



Refer to Table 31 for a list of the bit assignments for the interrupt flags field in the exchange package.

Table 31. Flag Register Bit Assignments

Word	Binary Exponent	Acronym	Name
12	31	RPE	Register Parity Error
12	30	MEU	Uncorrectable Memory Error
12	29	FPE	Floating-point Error
12	28	ORE	Operand Range Error
12	27	PRE	Program Range Error
12	26	EEX	Error Exit (000 issued)
12	25	BPI	Breakpoint Interrupt
12	24	MEC	Correctable Memory Error
12	23	MCU	MCU Interrupt
12	22	RTI	Real-time Interrupt
12	21	ICP	Interrupt from Internal CPU
12	20	IOI	I/O Interrupt (if IIO and SIE) <sup>†</sup>
12	19	PCI	Programmable Clock Interrupt
12	18	DL	Deadlock Interrupt
12	17	MII	001 $ij \neq 0$ or 033 Instruction Interrupt (if IMI and not MM)
12	16	NEX	Normal Exit (004 issued)
12	15	AMI	Address Multiply Interrupt

<sup>†</sup> SIE = System I/O interrupt enabled.

**VL** – vector length, word 13, bits 0 through 7

The VL register holds the content of the VL register. The 8-bit field contains the number of elements to be operated on in the vector register. In a CRAY T90 series system, if VL = 000 or VL = 200, all 200<sub>8</sub> vector elements are used within the vector register.

**XA** – exchange address, word 17, bits 16 through 31

The 16-bit field specifies the address of the first word of the next exchange package. This exchange package is loaded when any one of the following conditions occurs:

- An interrupt occurs that sets any of the following flags: RPE, MEU, FPE, OPR, BPI, MEC, MCU, RTI, ICP, IOI, PCI, DL, MII, NEX, or AMI
- A 000 is issued
- A 0040jk is issued with *k* being an illegal value (5, 6, or 7)

The XA field contains only bits 5 through 20. The lower bits are assumed to be 0's.

**EXIT Address 0 through 4**, words 15, 16, 17 bits 0 through 31

Each of the five 16-bit fields specifies the starting address of a 32-word exchange package. The *k* field of the 0040jk instruction specifies the exchange package to use. Only *k* fields equal to 0 through 4 are valid; if an invalid value is used, the exchange is to the XA address. Exit Address (EA) 0 is expected to be used for normal exits to maintain compatibility with existing systems.

Each EA field contains only bits 5 through 20. The lower bits are assumed to be 0's.

**CLN** – cluster number, word 13, bits 24 through 31

The CLN contains a 8-bit field. There are up to 36<sub>8</sub> clusters in the system, depending on the system configuration.

PPN – Processor number, word 13, bits 16 through 22

The contents of the 7-bit field in the exchange packages show the logical number of the CPU in which the exchange was executed. The maximum number is 127.

LATS – Words 0 through 17. Refer to the exchange package diagram for bit layouts.

Each LAT has four associated fields; Table 32 identifies those fields.

Table 32. LAT Fields

Field Name	Description
Logical Base	First logical address of this LAT
Logical Limit	Last address +1 of this LAT
Physical Bias	Physical bias = Physical base address – Logical base address
Modes	The controlling bits for each LAT R(ead), W(rite), X(ecute), C(achable), D(irty)

The use of LATs allows programs to share memory space. For example, two user jobs could reference the same library routine in memory while keeping their local code private.



# REAL-TIME CLOCK PROGRAMMABLE CLOCK INTERRUPT STATUS REGISTER PERFORMANCE MONITOR

Refer to the following subsections for information about the real-time clock, programmable clock interrupt, status register, and the performance monitor.

## Real-time Clock

---

A CRAY T90 series computer system contains one 64-bit real-time clock (RTC) per central processing unit (CPU). The RTC is synchronized when a CPU issues a 0014j0 instruction. The 0014j0 instruction causes all CPUs in the same cluster to be loaded with the contents of Sj. The RTC is located on two HD options, each of which handles 32 bits. The HD000 option handles bits 0 through 31; the HD001 option handles bits 32 through 63.

HD000 will detect a carry, out of the RTC, at a count of 3777777776 during normal operation. HD001 then increments the upper bits during the next clock period, and HD000 suppresses any toggles.

The RTC is incremented once every clock period. The RTC allows for clock-period timing of program execution. When the machine is deadstarted, the RTC must be loaded in order to synchronize all the CPUs. If they are not synchronized, each CPU will have a different RTC value.

Writing to the RTC with the 0014j0 instruction sends a copy of the Sj register from the CPU issuing the instruction to all RTC registers via the issue paths of the shared registers. Reading the RTC with a 072i00 instruction copies the RTC register of the CPU that issued the 072i00 instruction into the scalar registers.

Refer to Figure 100 for an RTC and programmable clock interrupt (PCI) block diagram.

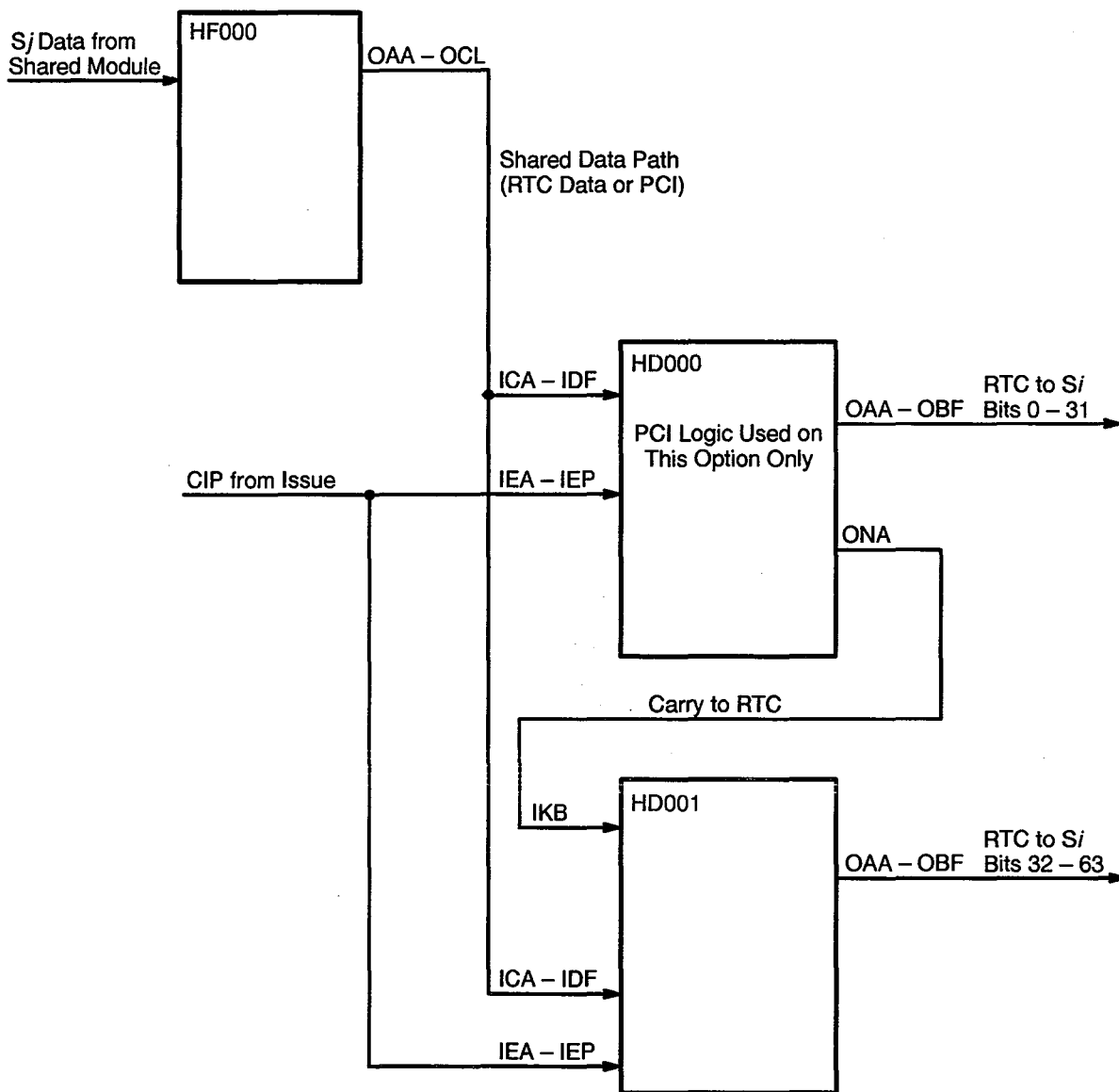


Figure 100. RTC and PCI Block Diagram

## Programmable Clock

Each CPU has one programmable clock (PC), which is a 32-bit counter. The programmable clock decrements every clock period; the clock is located on the HD000 option.

The programmable clock is loaded by the 0014j4 instruction when the program is in monitor mode. When the programmable clock equals zero, an interrupt request (PCI) is generated. To generate a PCI, the IPC mode bit must be set. In user mode, IPC must have been set in the user's exchange package. If the CPU is in monitor mode, either IPC was set in

the monitor's exchange package, or a 001406 instruction was issued. The interrupt request remains set until cleared by a 001405 instruction. If the CPU is in monitor mode, and if the interrupt request is not desired, use a 001407 instruction to disable the IPC mode bit.

The PCI request is enabled and disabled on the HD option, which contains the exchange parameters.

## RTC and PC Instructions

---

Refer to Table 33 for a list of the RTC and PC instructions.

Table 33. RTC and PC Instructions

Instruction	CAL	Description
0014j0 †	RT Sj	Enter RTC register with Sj
072i00	Si RT	Transmit RTC to Si
0014j4 †	PCI Sj	Transmit Sj to programmable clock
001405 †	CCI	Clear PCI request
001406 †	ECl	Enable PCI request
001407 †	DCI	Disable PCI request

† Data cache is a monitor mode instruction.

## Performance Monitor

---

The performance monitor (PM) is normally used to monitor software performance. With the results of the performance monitor, a programmer can determine how efficiently a program is running in the system. If, for example, the program is performing too many instruction fetches or too many hold issue conditions are occurring, the programmer can review the program structure and modify it to minimize these occurrences.

Each CPU contains a performance monitor; because each CPU is identical, all references in this section pertain to a single CPU. Each CPU contains 32 performance counters and each counter is 48 bits wide. Table 34 shows which event each counter monitors. Each counter increments each time a particular event occurs in the CPU while the CPU is in nonmonitor mode (IMI bit is not set). The counters related to memory references may be incremented by as many as eight times per clock period (CP). Counters related to vector operations are incremented by the value in the vector length register at the time the instruction issues.

Table 34. Performance Monitor

Counter	Event Monitored	Instructions	Increments
0	Number of: Clock periods monitored		+1
1	Instructions issued		+1
2	Clock periods holding issue		+1
3	Instruction fetches		+1
4	CPU memory references (ports A, B, C)		+8
5	Clock periods for references (ports A, B,C)		+2047
6	I/O memory references (port D, I/O only)		+2
7	Cache misses		+1
10	Holding issue on: A registers and access conflicts		+1
11	S registers and access conflicts		+1
12	V registers		+1
13	B/T registers		+1
14	Functional units		+1
15	Shared registers		+1
16	Memory ports		+1
17	Number of cache hits		+1
20	Number of instructions: Instructions 000000 through 004000	000 – 004	+1
21	Branches	005 – 017	+1
22	Address instructions	02x, 030 – 033, EIS 042 – 057 ,073i20, 073i30	+1
23	B/T memory instructions	034 – 037	+1
24	Scalar instructions	040 – 043, 071 – 077 except 073i20, 073i30	+1
25	Scalar integer instructions	044 – 061, 070ij6	+1
26	Scalar floating-point instructions	062 – 070	+1
27	S/A memory instructions	10x – 13x	+1
30	Number of operations: Vector logical	070ij1, 140 – 147, 1740j4 – 1740j6, 175	+VL
31	Vector shifts, pop., leading zero	150 – 153, 174xx (1 – 3)	+VL
32	Vector integer adds	154 – 157	+VL
33	Vector floating-point multiplies	160 – 167	+VL
34	Vector floating-point adds	170 – 173	+VL
35	Vector floating-point reciprocals	174xx0	+VL
36	Vector memory reads	176	+VL
37	Vector memory writes	177	+VL



## Performance Monitor Instructions

---

Table 35 lists all the instructions associated with the performance monitor.

Table 35. Performance Monitor Instructions

Instruction	CAL	Description
001500		Clear all performance counters
073 <i>ij</i> 1	<i>Si</i> <i>SRj</i>	Transmit ( <i>SRj</i> ) to <i>Si</i> (monitor mode only for $j = 2 - 7$ )
073 <i>i</i> 05	SR0 <i>Si</i>	Transmit ( <i>Si</i> ) bits 48 – 52 to SR0
073 <i>i</i> 25	SR2 <i>Si</i>	Advance performance monitor pointer
073 <i>i</i> 75	SR7 <i>Si</i>	Transmit ( <i>Si</i> ) to maintenance channel

## Clearing the Performance Counters

---

Instruction 001500 clears all performance counters. This instruction must be issued while the CPU is in monitor mode in order for the instruction to operate correctly.

## Reading the Performance Monitor

---

The performance monitor is read with the 073*i*21 and 073*i*31 instructions. Each counter is read 48 bits at a time and requires that two instructions be issued to read all the counters. The 48 bits of the counter read are stored in the *Si* register. When the 073*i*21 instruction is issued, counters 0 through 17 are sent to *Si*. The 073*i*31 instruction, when issued, reads counters 20 through 37 and sends the bits to *Si*.

The system hardware requires a minimum of 3 CPs between issuing 073*ix*1 instructions. Also, the PM Busy Status (PMBY) bit (bit 47 of SR0) must be cleared before reading the counters. If the 3-CP wait is not written into the program, an undeterminable corruption of performance monitor data occurs.

## Performance Monitor Block Diagram

---

Refer to Figure 101 for the performance monitor block diagram. The performance monitor is composed of the HF000, HD000, and HD001 options. The HF000 option contains the lower bits (0 through 31) and the HD000 and HD001 options contain the upper bits (32 through 47) for all 32 counters; there is one counter for each event tracked by the performance monitor. These 48-bit counters are incremented as each event occurs, as long as the CPU is not in monitor mode.

## Status Register

---

A CRAY T90 series computer system has eight status registers, which are located on the HD and HF options. The status register is no longer part of the exchange package as it was in previous systems. Figure 102 shows the status register format and bit assignments of each register. The status registers are read by the 073ij1 instruction.

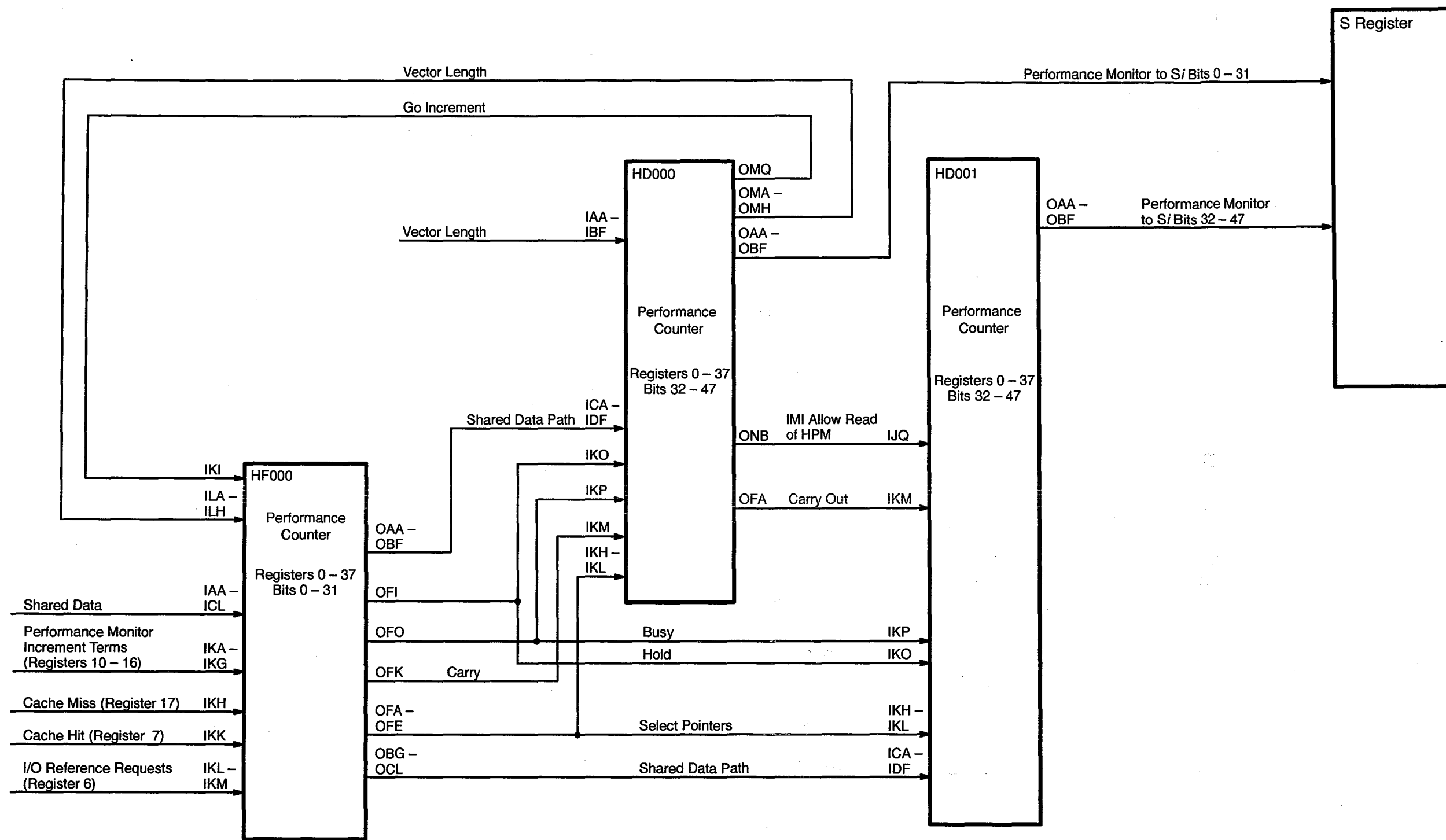


Figure 101. Performance Monitor Block Diagram

Bits 63	57	52	48	47	40	39	32	31	16	15	0			
SR0	C L N ≠0	B M L	I B P S	F P F O D P R M	I B P M B Y	Processor Number 6	Cluster Number 0	7	0					
SR1														
SR2														
SR3														
SR4														
SR5														
SR6														
SR7														
Bits 63	62	61	55	54	48	47	46	43	32	31	24	16	15	0

† SR0 bit 20 = monitor mode · maintenance mode · not (SR7 busy)

Figure 102. Status Registers

The eight status registers are further defined in Table 36 through Table 40.

Status register 0 (SR0) shows the status of several bits in the active exchange package.

Table 36. Status Register (SR0)

Bits	Name	Description
63	CLN≠0	Cluster number not equal to zero
57	BML	Bit matrix loaded
52	IBP †	Interrupt on breakpoint
51	FPS †	Floating-point status
50	IFP †	Interrupt on floating-point error
49	IOR †	Interrupt on operand range error
48	BDM †	Bidirectional memory
47	PMBY	Performance monitor busy
40 through 43	PN	Processor number
32 through 39	CLN	Cluster number

† Designates that this was written by a 073/05 instruction. All other bits of SR0 are read-only.

Status register 1 (SR1) is not defined.

Status register 2 (SR2) bits 0 through 47 are bits of the performance monitor counters 0 through 17.

Status register 3 (SR3) bits 0 through 47 are bits of the performance monitor counters 20 through 37.

Status register 4 (SR4) bits are shown in Table 37. SR4 contains the correctable and uncorrectable memory error flags, port bits, and read mode bits. The error information stored in SR4 is latched into the register and held until the register is read. Once SR4 is read, the register is cleared, and new error data can be stored in the register. If multiple errors occur, only the first error is held in SR4. Bits 32 through 45 define the destination code associated with the error. Table 37 is a decode of these destination bits.

Table 37. Status Register 4 (SR4)

Bits	Name	Description
47	UME	Uncorrectable memory error
46	GME	Correctable memory error
32 through 45	CODE	Destination code (refer to Table 38)

Table 38. Destination Codes

Destination	Bit														
	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Cache read	1	1	1	–	Word										
V register read	1	1	0	Register	–	Element									
S register read	1	0	1	Register	0	–									
A register read	1	0	1	Register	1	–									
T register read	1	0	0	–	0	–	Register								
B register read	1	0	0	–	1	–	Register								
Fetch read	0	1	1	Group	–					Word					
I/O read	0	1	0	Type	–					Word					
Exchange read	0	0	1	–					Word						
I/O write	0	0	0	Type	1	–									
Processor write	0	0	0	–	0	1	0	A/S	–						
Reconfigure	0	0	0	–	1	1	0	–							
Memory error	0	0	0	–	0	0	0	–							

Status register 5 (SR5) bits 32 through 43 contain the syndrome code of the memory error. The information is held until the status register is read.

Status register 6 (SR6) bits 32 through 44 contain the error address for the memory error. These bits are latched into the SR6 on a memory error. The information is held until the status register is read.

Status register 7 (SR7) contains information on LAT faults, register parity errors (RPE), and shared register errors (SRRE). Bits 48 through 54 contain an LAT miss flag for each memory port. Bits 55 through 61 contain an LAT multiple-hit flag for each memory port. Bit 47 is the RPE

flag. If this bit sets, then bits 32 through 43 contain the chip number. Bit 46 is the SRRE flag, and if this flag is set, bits 24 through 31 contain the chip number.

Table 39. Status Register 7 Bit Definitions

Bits	Name	Description
48 through 54	LAT fault	LAT miss
55 through 61	LAT fault	Multiple LAT hit
46	SRRE	Shared register read error
24 through 31		Shared register chip number
47	RPE	Register parity error
32 through 43		RPE chip number

Table 40. Register Parity Error Code

Octal	Option	Description
001 000	VR0	Vector register V0 pipe 0
001 001	VR1	Vector register V1 pipe 0
001 010	VR2	Vector register V2 pipe 0
001 011	VR3	Vector register V3 pipe 0
001 100	VR4	Vector register V4 pipe 0
001 101	VR5	Vector register V5 pipe 0
001 110	VR6	Vector register V6 pipe 0
001 111	VR7	Vector register V7 pipe 0
010 000	VR8	Vector register V0 pipe 1
010 001	VR9	Vector register V1 pipe 1
010 010	VR10	Vector register V2 pipe 1
010 011	VR11	Vector register V3 pipe 1
010 100	VR12	Vector register V4 pipe 1
010 101	VR13	Vector register V5 pipe 1
010 110	VR14	Vector register V6 pipe 1
010 111	VR15	Vector register V7 pipe 1
011 000	CH0	Data cache bits 0 – 3, 32 – 35 Sect. 0,1,6,7
011 001	CH1	Data cache bits 0 – 3, 32 – 35 Sect. 2,3,4,5
011 010	CH2	Data cache bits 4 – 7, 36 – 39 Sect. 0,1,6,7







# SCALAR CACHE

Each CPU has a scalar data cache. The cache accelerates common memory data access for address register and scalar register read requests. Only address and scalar registers can access the cache.

The data cache has the following features:

- The cache is organized into 8 pages of data. Each page contains 8 lines of 16 words, thus providing 1,024 words of data in the cache. Figure 103 illustrates the logical layout of the cache.
- Cache is parity protected; each 8-bit byte has an associated parity bit. If enabled, a parity error on a cache read will cause an interrupt.
- When an A or S register memory reference is made, one of two things may occur: a *cache hit* or a *cache miss*.
- A and S register store requests are *write-through*. The cache word will be updated if there is a hit; if a miss occurs, no cache lines are requested.
- B, T, and V register store requests cause corresponding cache lines to be set invalid on a cache hit. Store requests on a cache miss have no effect on the cache. B, T, and V register load requests also have no effect on the cache.

## Cache Hit

---

A cache hit is determined using logical addresses, not physical addresses. A cache hit occurs when the following conditions are met.

- A valid page address consisting of address bits 7 through 39, held within the cache, matches the corresponding address bits of a memory request.
- The cache line indicated by bits 4 through 6 of the requesting address is valid within the cache.

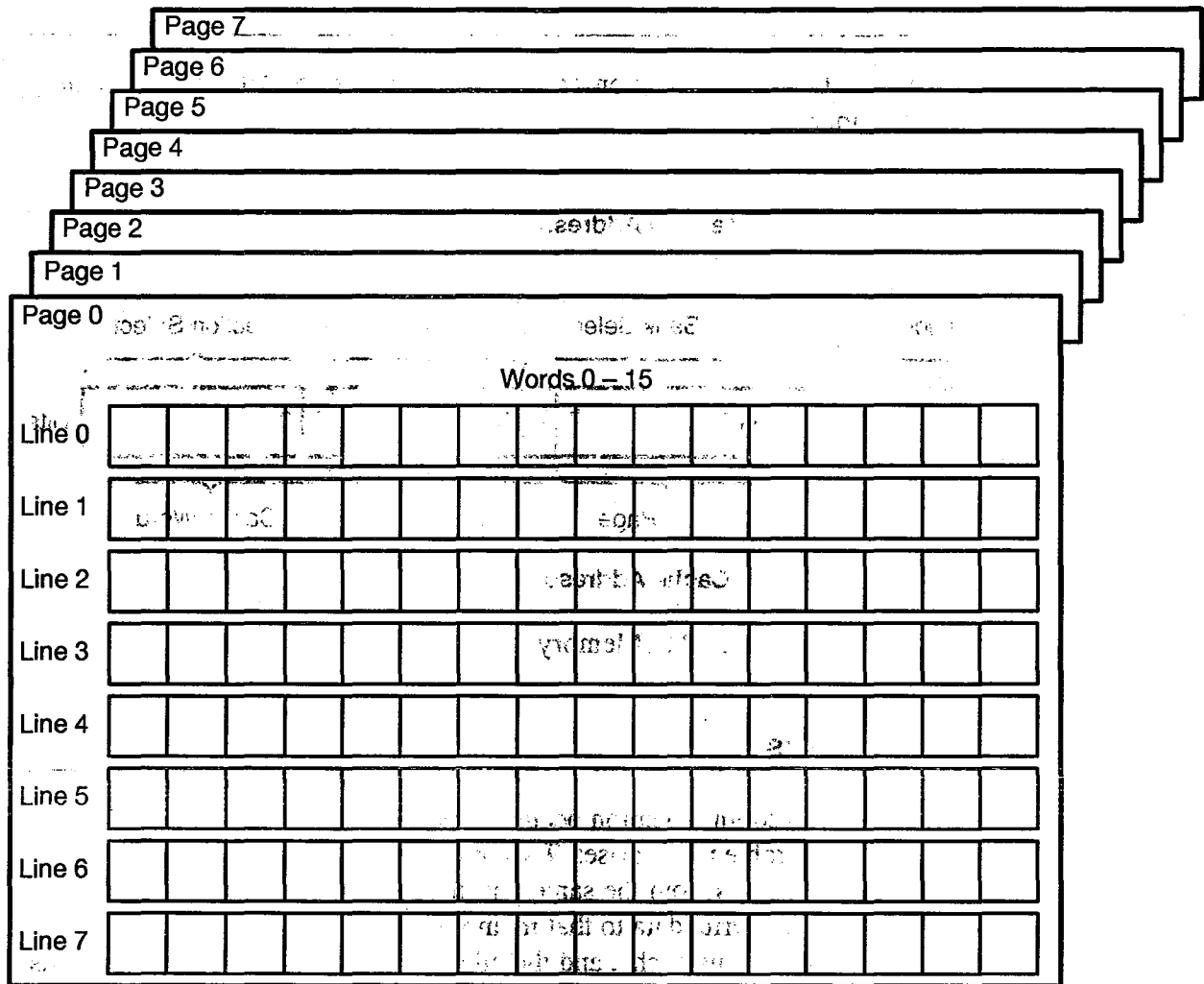


Figure 103. Cache Layout

### Cache Miss

A cache miss occurs when a request from an A or S register load request does not match a page address. When this occurs, the corresponding line is requested from memory and the previously valid page address is set to the new page address. All lines in the new page are set invalid. As the new requested line returns from memory, the new page address is set valid as is the cache line that was requested.

Another type of miss occurs when a memory reference matches the page but not any line in the page, or the page is not valid. When this occurs, 16 sequential words are requested from memory, and the line is set valid.

## Cache Addressing

Figure 104 shows how memory addresses are used to determine a cache hit or miss.

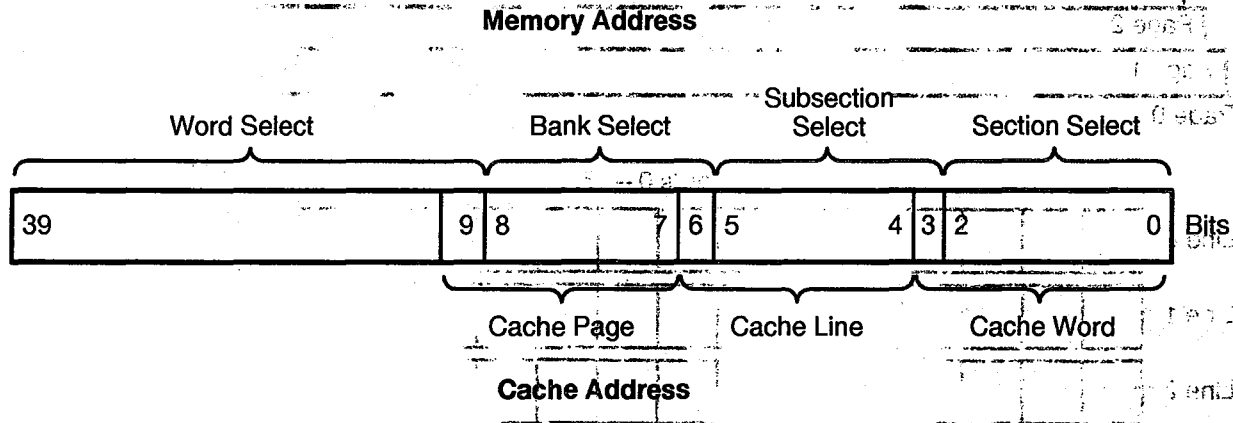


Figure 104. Memory Addresses

## Potential Cache Problems

Because no communication occurs between caches in different CPUs, the following problem can arise: Two or more CPUs can have data in their respective caches from the same physical address in memory, and one of the CPUs can write data to that memory address. The CPU that wrote the data will update its cache, and the other CPUs will contain old data. This problem can be managed in several ways:

- There are load instructions that bypass cache. These instructions cause the cache line to be invalidated on a cache hit.
- LATs can be set up to define areas of memory that are not cache enabled.
- If the SCE (scalar cache enable) bit is not set in the exchange package, it will prevent the use of cache for that job.

Another problem that can occur is when you go through memory with a stride value of 128; this causes memory to *thrash*. A stride of 128 will use 1 word of 1 line from each cache page; then when you start replacing lines, you will get 16 words back from memory to cache but will be using only 1 word. This problem can be avoided by redesigning user code.

## CH Option

There are 16 CH options; these options contain all of the cache memory RAMs. The even-numbered CHs hold data from memory sections 0, 1, 6, and 7; the odd-numbered CHs hold data from memory sections 2, 3, 4, and 5.

On a memory write, each CH writes 4 bits to all memory sections. Table 41 shows the bits per option.

Table 41. CH Option Bits

	CH000	CH002	CH004	CH006	CH008	CH010	CH012	CH014
Read Data Sect 0,1,6,7	0-3 32-35	4-7 36-39	8-11 40-43	12-15 44-47	16-19 48-51	20-23 52-55	24-27 56-59	28-31 60-63
Write Data Sect. 0-7	0-3 CB 0	4-7 CB 1	8-11 CB 2	12-15 CB 3	16-19 CB 4	20-23 CB 5	24-27 CB 6	28-31 CB 7
	CH001	CH003	CH005	CH007	CH009	CH011	CH013	CH015
Read Data Sect 2,3,4, 5	0-3 32-35	4-7 36-39	8-11 40-43	12-15 44-47	16-19 48-51	20-23 52-55	24-27 56-59	28-31 60-63
Write Data Sect. 0-7	32-35 CB 8	36-39 CB 9	40-43 CB 10	44-47 CB 11	48-51	52-55	56-59	60-63

## Scalar Cache Instructions

Refer to Table 42 for a list of the scalar cache instructions.

Table 42. Scalar Cache Instructions

Instruction	CAL	Description
002501	ESC	Enable scalar cache
002601	DSC	Disable and invalidate scalar cache
10h20mn	Ai exp,Ah,BC	Load Ai from ((Ah)+exp) bypassing data cache and invalidating cache line
10h60pmn	Ai exp,Ah,BC	Load Ai from ((Ah)+exp) bypassing data cache and invalidating cache line
12h20mn	Si exp,Ah,BC	Load Si from ((Ah)+exp) bypassing data cache and invalidating cache line
12h60pmp	Si exp,Ah,BC	Load Si from ((Ah)+exp) bypassing data cache and invalidating cache line