CPU MODULE (CPE1)

Figures

Tables

CPU MODULE (CPE1)

CPE1 General Description

The CPE1 module contains the central processing unit (CPU) for the CRAY T90 series computer systems. There is one CPU per CPE1 module. This CPU uses the IEEE standard format for floating-point arithmetic.

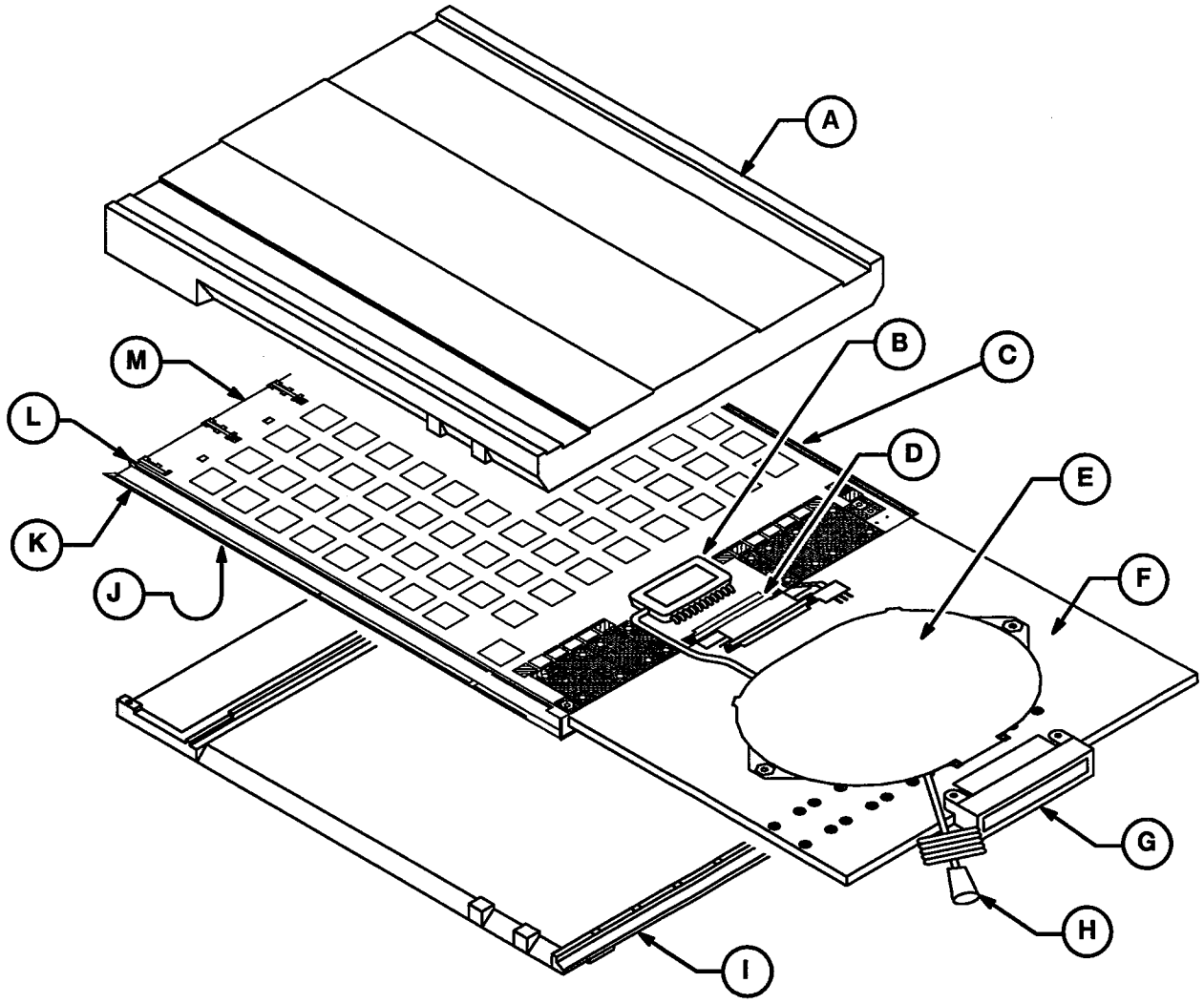
There have been many enhancements to the CRAY T90 series CPU, and several new instructions have been added to increase the performance. Figure 1 illustrates CP module components. Figure 2 and Figure 3 show the basic functions and locations of all options on a CP module. Figure 4 shows a block diagram 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 systems contains up to two stacks of as many as eight modules. A CRAY T932 system contains up to four stacks of as many as eight modules.

Each module in a stack functions independently; 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. The 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 Flt Mult	RC000 Recip	TZ000 Clock	HM000 Logic Monitor	MZ000 BS Fanout	TW010 Not Used	RC001 Recip	NA001 Flt Mult
TW000 Not Used	NC000 Flt Mult	RB000 Recip	FA000 Flt Add Coeff	TW006 Not Used	FA001 Flt Add Coeff	OA002 BMM and Parity	RB001 Recip	NC001 Flt 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 Check-bit 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
ZB008		ZB000		ZB004		ZB002		ZB006

Figure 2. Option Layout Board 1

								HC000 I/O Relay Data
ND001 Flt Mult	AM001 Integer Multi	TW011 Not Used		HM001 Logic Monitor		AM002 Integer Multi	ND000 Flt Mult	
NB001 Flt Mult	RA001 Recip	OA005 BMM and Parity	FB001 Flt Add Exponent	TW007 Not Used	FB000 Flt Add Exponent	RA000 Recip	NB000 Flt 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 Check-bit 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 SECDED
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
ZB007		ZB003		ZB005		ZB001		ZB009

Figure 3. Option Layout Board 2

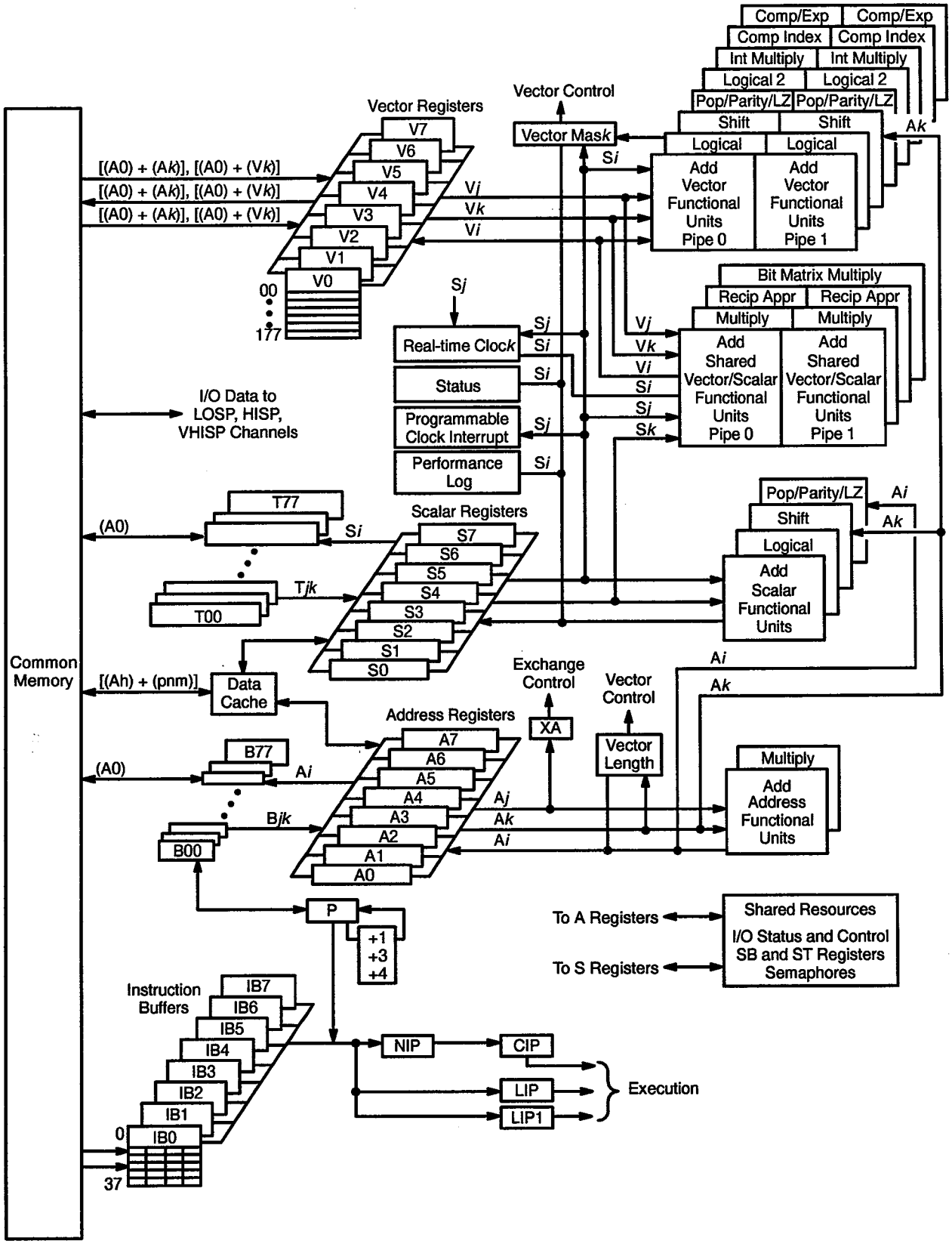


Figure 4. CPU Block Diagram

ADDRESS AND SCALAR REGISTERS

The address and scalar registers are located together 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 AV option, three AW options, two AX options, and two AY 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 illustrated in Figure 5, each AV000, AW000, AW001, AW002, AX000, AX001, AY000, and AY001 option contains an 8-bit slice of the address registers. Figure 5 also illustrates the input and output data paths for the address and scalar registers.

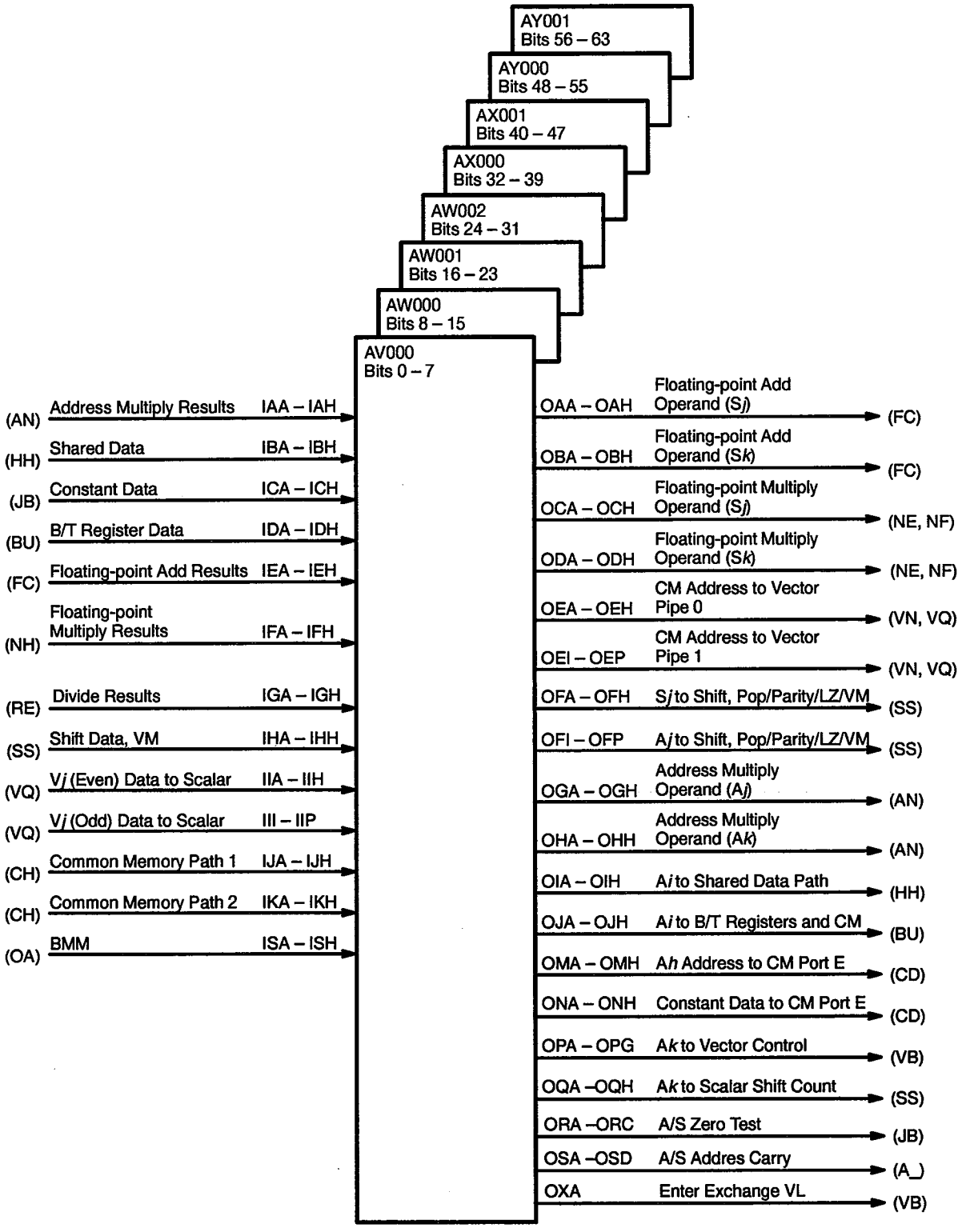


Figure 5. Address and Scalar Register Data Paths

Entry Codes

During the instruction decode on the JB option, the A/S register options receive an A/S entry code from the the JB option. This code generates the control that is 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* Constants
1	023*ij*0 S*j*
2	023*ij*1 VL data
3	024*ijk* B data
4	030,031*ijk* Add
5	026*ij* (0 – 3), 027*ij* (0 – 1) pop/par/lz
6	032*ijk* A multiply
7	022*ijk*, 04 (2 – 3) *jk*/mask data
10	N/A
11	073*i* (2 – 3) 0 VM data
12	N/A
13	N/A
14	04 (4 – 7) *ijk*, 05 (0 – 1) *ijk* Logical
15	N/A
16	05 (2 – 5) *ijk*, 05 (6 – 7) *ijk* Shift
17	N/A

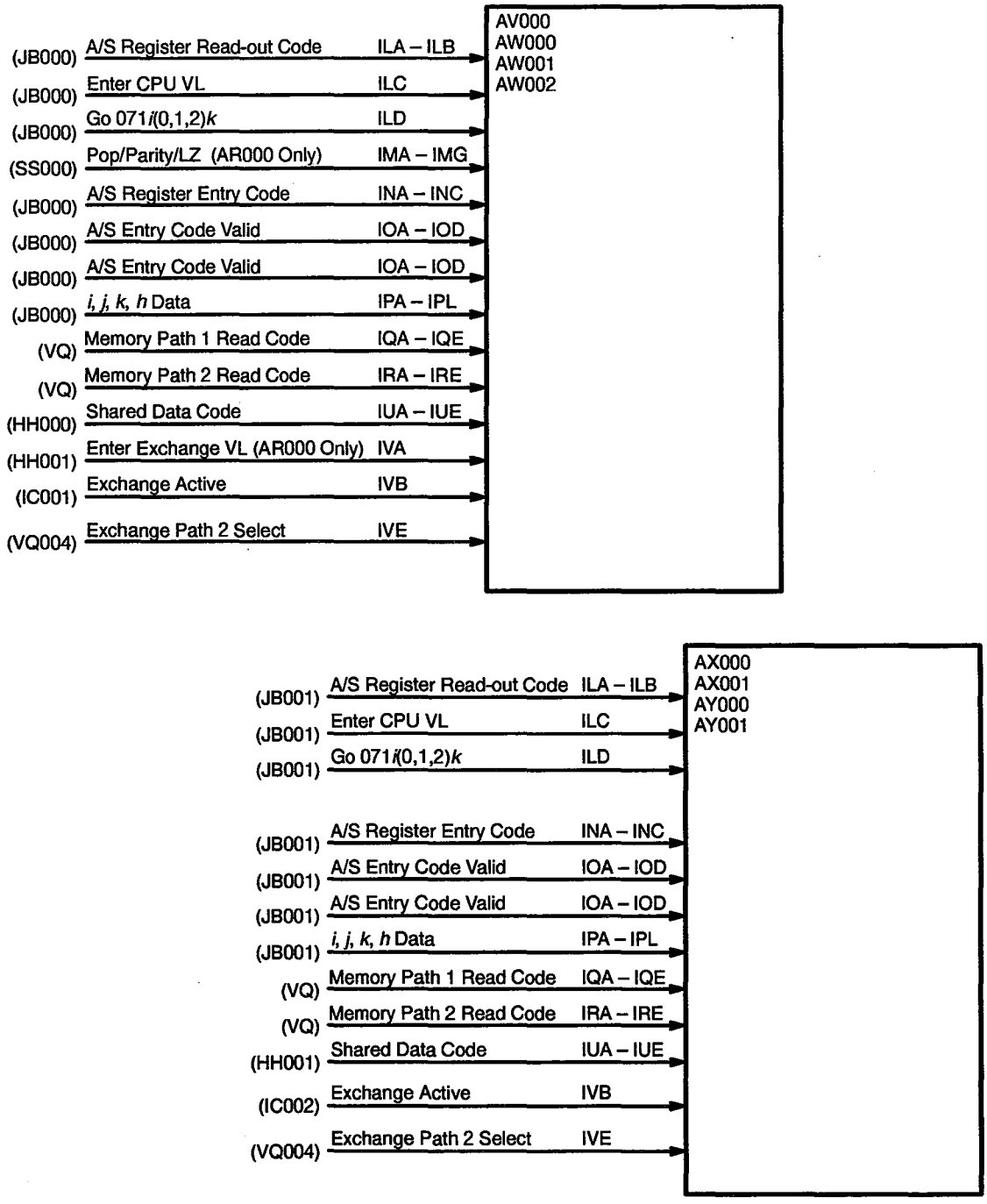


Figure 6. A/S Control Terms

A Register Memory References

Refer to Figure 7 for a memory-to-A/S-register block diagram. The address registers read or write one word of memory during each instruction. The B registers provide intermediate storage for the address registers and perform memory block references; one B register instruction can access a group of operands from memory. The A registers use these operands to generate results that are sent back to the B registers and stored in memory. Using the B registers as buffer storage, a block reference requires fewer clock periods than if several individual address or scalar references were issued.

The A registers also have an access path to cache memory, which 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 from 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 CRAY T90 series systems are 64 bits wide, special 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, enabling A registers to be shifted and logical operations to be performed. To execute these special mode instructions, an EIS 005400 instruction must precede the actual A register instruction.

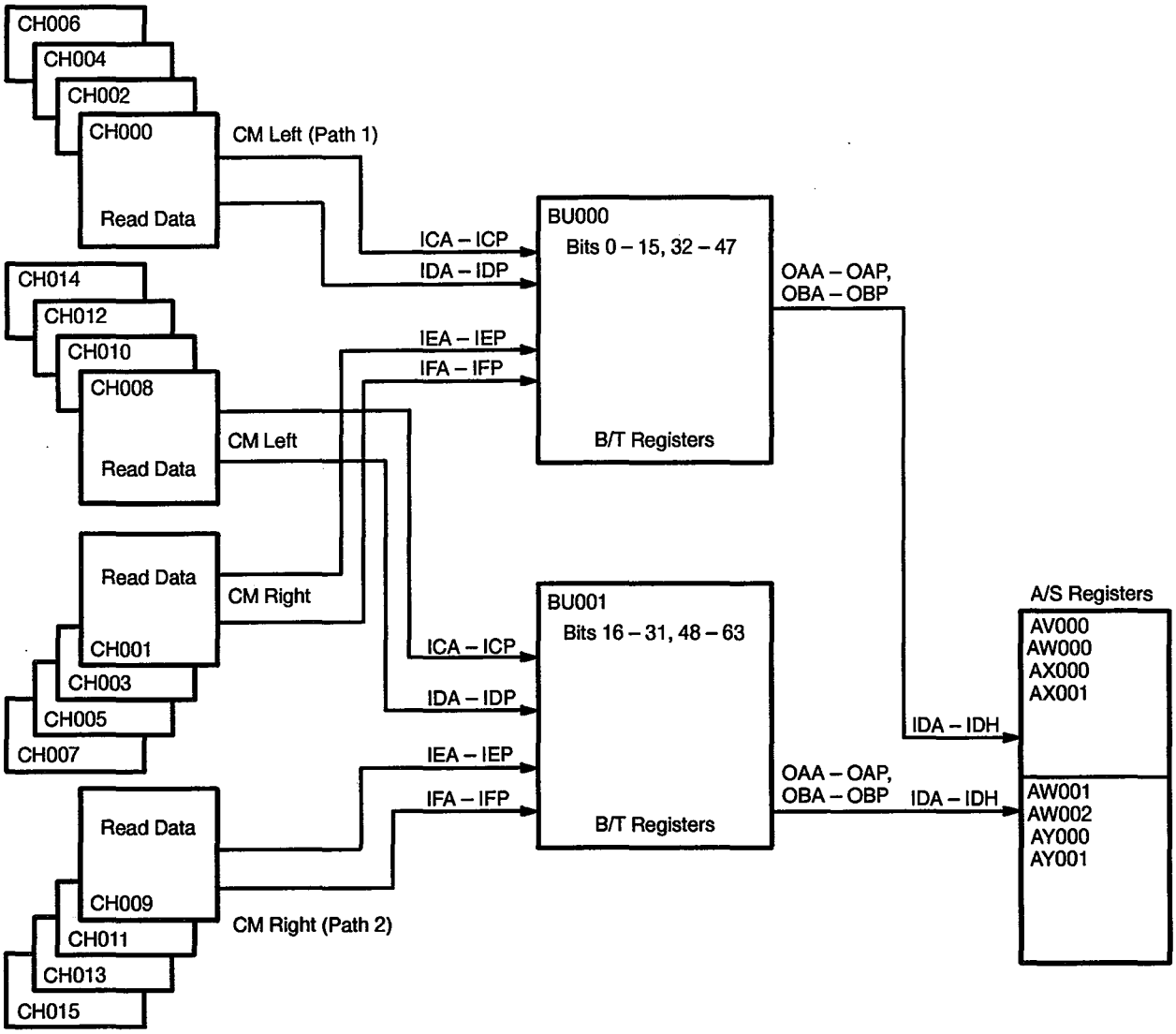


Figure 7. Memory-to-A/S Register Block Diagram

Scalar Registers

The CPU contains eight 64-bit scalar registers that are designated S0 through S7. The scalar registers are contained on the AV, AW, AX, and AY options (refer to Figure 5).

The scalar registers send operands to and receive results from the scalar functional units and the floating-point functional units. The functional units perform integer and floating-point arithmetic and logical operations. The scalar registers read and write central memory through the T registers, read and write the data cache, and provide 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 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

Scalar registers read or write one word of memory during each instruction. The T registers provide intermediate storage for the scalar registers, and can perform memory block references; a single instruction can access a group of operands from memory. These operands are then used by the scalar registers to generate results that can be sent back to the T registers and stored in memory. Using the T registers as buffer storage, a block reference requires fewer clock periods than if several individual address or scalar references were issued.

The S registers also have an access path to cache memory, which 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, a value of 0 is sent out regardless of the actual value stored in S0. When the *k* field equals 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.

- 040*i*00 *nm* S*i* *exp*: loads the quantity *nm* into the lower 32 bits of register S*i*. The upper 32 bits are cleared.
- 041*i*00 *nm* S*i* *exp*: loads the one's complement of *nm* into the lower 32 bits of register S*i*. The upper 32 bits are all 1's.
- 040*i*20 *nm* S*i* *exp*: loads the quantity *nm* into the lower 32 bits of register S*i*. The upper 32 bits are unchanged.
- 040*i*40 S*i* *exp*: loads the quantity *nm* into the upper 32 bits of register S*i*. The lower 32 bits are unchanged.

B AND T REGISTERS

Each CPU contains 64 (100_8) 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 is 64 bits wide.

Two BU options, BU000 and BU001, compose the B and T registers. Each option contains 32 bits of each register. BU000 contains bits 00 through 15 and bits 32 through 47. BU001 contains bits 16 through 31 and bits 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.

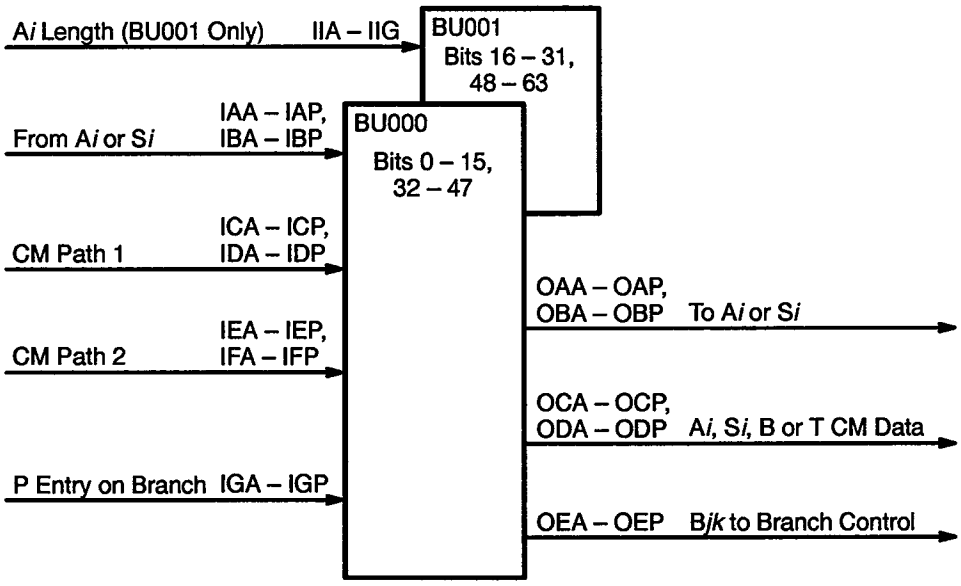


Figure 8. B and T Register Inputs and Outputs

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
0050*jk*	J B*jk*	Jump to B*jk*
0051*jk**	JINV B*jk*	Jump to B*jk* (invalidate instruction buffers)
024*ijk*	A*i* B*jk*	Transmit (B*jk*) to A*i*
025*ijk*	B*jk* A*i*	Transmit (A*i*) to B*jk*
034*ijk*	B*jk* A*i*, A0	Transmit (A*i*) words from common memory starting at address (A0) to B registers starting at register *jk*
035*ijk*	,A0 B*jk*,A*i*	Transmit (A*i*) words from B registers starting at register *jk* to memory starting at address (A0)
036*ijk*	T*jk* A*i*, A0	Transmit (A*i*) words from memory starting at address (A0) to T register starting at register *jk*
037*ijk*	,A0 T*jk*,A*i*	Transmit (A*i*) words from T registers starting at register *jk* to memory starting at address (A0)
074*ijk*	S*i* T*jk*	Transmit (T*jk*) to S*i*
075*ijk*	T*jk* S*i*	Transmit (S*i*) to T*jk*

* Denotes a maintenance mode instruction only.

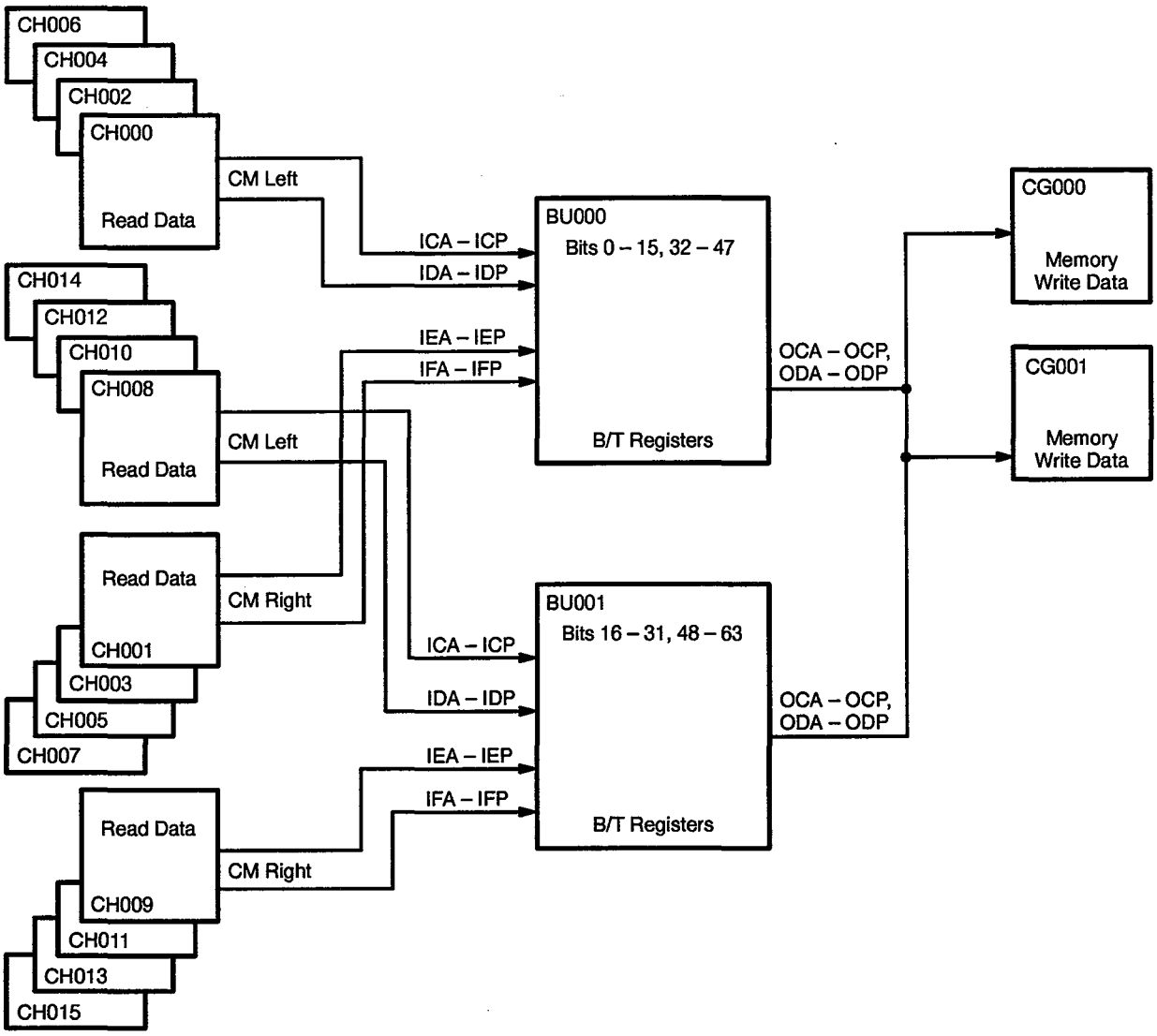


Figure 9. B/T-register-to-memory Block Diagram

ADDRESS AND SCALAR ADD

The address and scalar registers are contained on eight options: one AV option, three AW options, two AX options, and two AY 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
030*ijk*	A*i* A*j*+A*k*	Transmit integer sum of (A*j*) and (A*k*) to A*i*
030*i*0*k*	A*i* A*k*[S]	Transmit (A*k*) to A*i*
030*ij*0	A*i* A*j*+1[S]	Transmit integer sum of (A*j*) and 1 to A*i*
031*ijk*	A*i* A*j*–A*k*	Transmit integer difference of (A*j*) and (A*k*) to A*i*
031*i*0*k*	A*i* –A*k*[S]	Transmit inverse of (A*k*) to A*i*
031*ij*0	A*i* A*j*–*1*[S]	Transmit integer difference of (A*j*) and 1 to A*i*
060*ijk*	S*i* S*j*+S*k*	Transmit integer sum of (S*j*) and (S*k*) to S*i*
061*ijk*	S*i* S*j*–S*k*	Transmit integer difference of (S*j*) and (S*k*) to S*i*
061*i*0*k*	S*i* –S*k*	Transmit inverse of (S*k*) to S*i*

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.

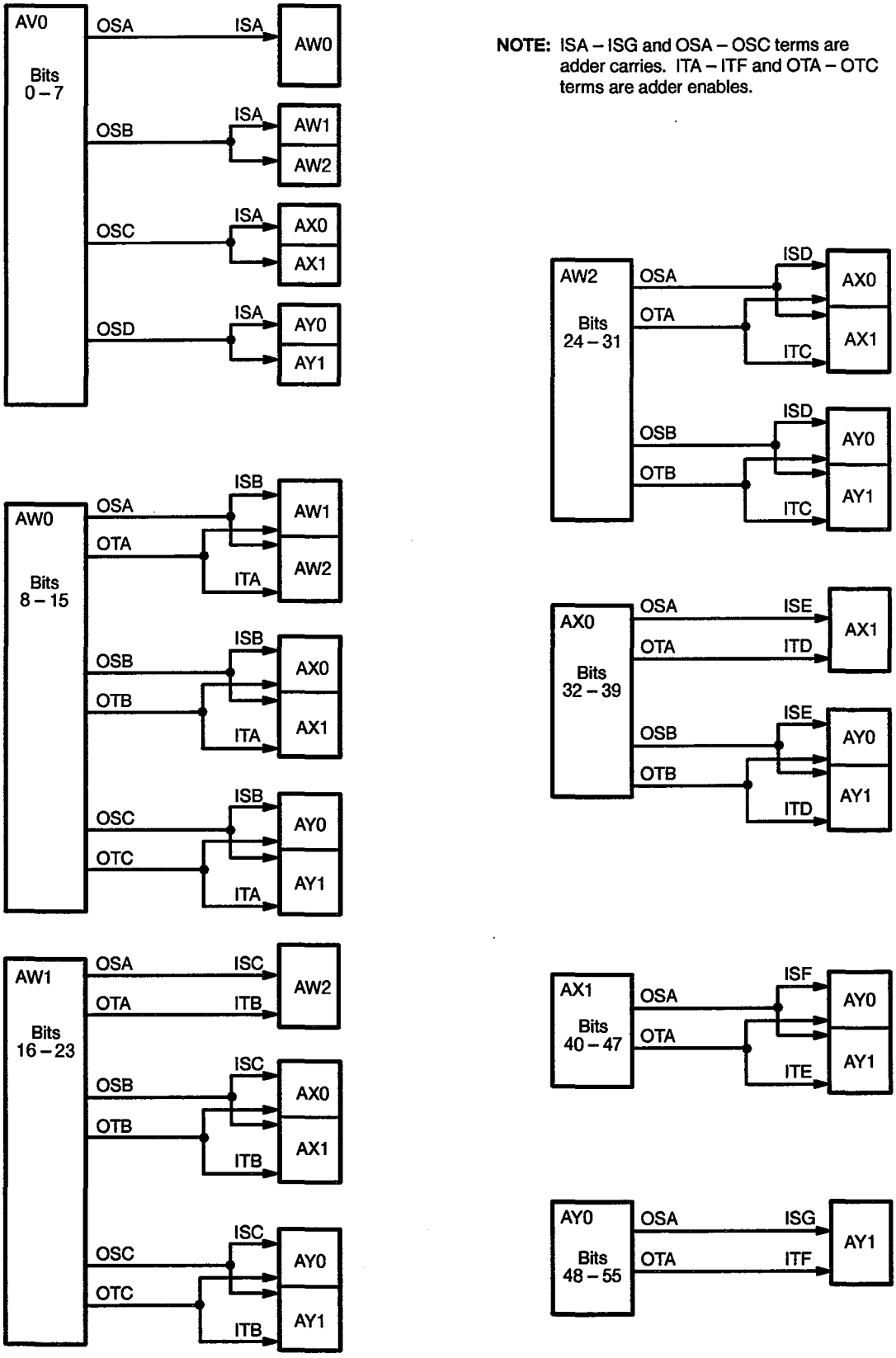


Figure 10. Carry Bit and Enable Bit Fanouts

SCALAR LOGICAL

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

Refer to Figure 11 for an illustration of the address and scalar registers. The scalar registers are contained on eight options: one AV option, three AW options, two AX options, and two AY 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.

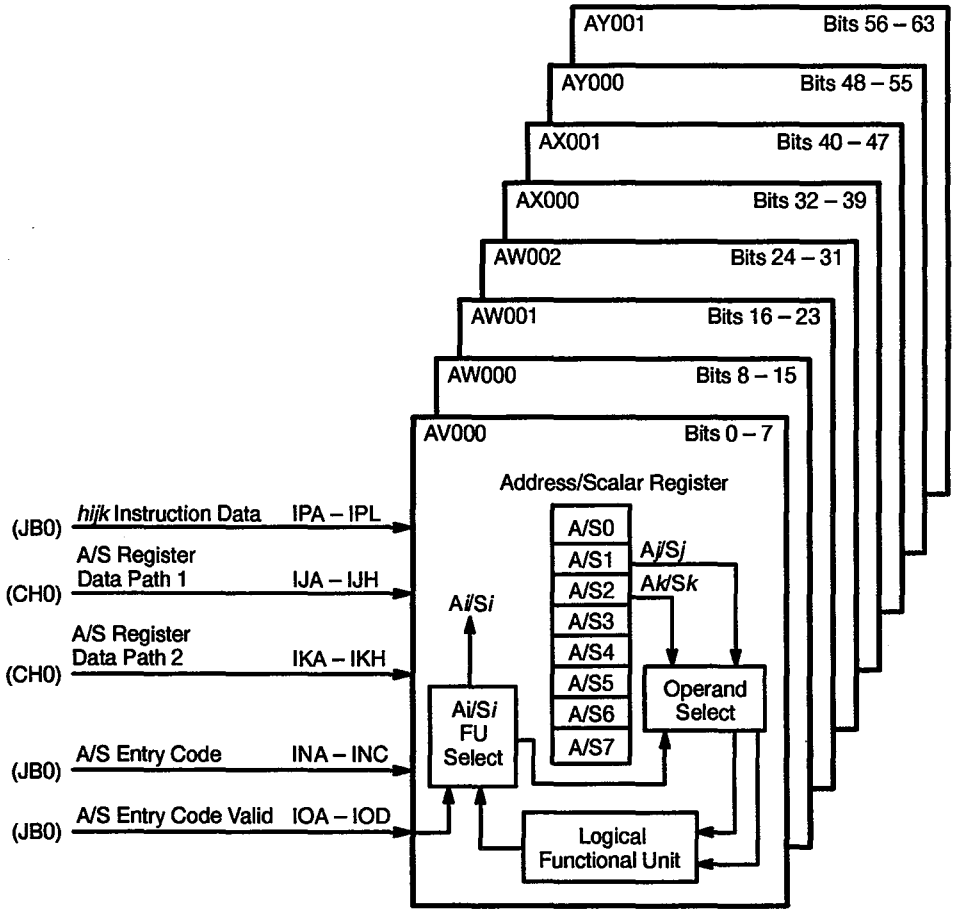


Figure 11. Address/Scalar Logical Block Diagram (Instructions 044*ijk* through 051*ijk*)

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.

Table 4. Scalar Logical Functional Unit Instructions

Instruction	CAL	Description
044*ijk*	S*i* S*j*&S*k*	Logical product of (S*j*) and (S*k*) to S*i*
044*ij*0	S*i* S*j*&SB	Sign bit of (S*j*) to S*i*
044*ij*0	S*i* SB&S*j*	Sign bit of (S*j*) to S*i* (S*j* ≠ 0)
045*ijk*	S*i* #S*k*&S*j*	Logical product of (S*j*) and one's complement of (S*k*) to S*i*
045*ij*0	S*i* #SB&S*j*	(S*j*) with sign bit cleared to S*i*
046*ijk*	S*i* S*j*\S*k*	Logical difference of (S*j*) and (S*k*) to S*i* (S*j* ≠ 0)
046*ij*0	S*i* S*j*\SB	Transmit (S*j*) with sign bit toggled to S*i*
046*ij*0	S*i* SB\S*j*	Transmit (S*j*) with sign bit toggled to S*i* (S*j* ≠ 0)
047*ijk*	S*i* #S*j*\S*k*	Logical equivalence of (S*k*) and (S*j*) to S*i*
047*i*0*k*	S*i* #S*k*	Transmit one's complement of (S*k*) to S*i*
047*ij*0	S*i* #S*j*\SB	Logical equivalence of (S*j*) and sign bit to S*i*
047*ij*0	S*i* #SB\S*j*	Logical equivalence of (S*j*) and sign bit to S*i* (S*j* ≠ 0)
047*i*00	S*i* #SB	Enter one's complement of sign bit into S*i*
050*ijk*	S*i* S*j*!S*i*&S*k*	Logical product of (S*i*) and (S*k*) complement ORed with logical product of (S*j*) and (S*k*)
050*ij*0	S*i* S*j*!S*i*&SB	Scalar merge of (S*i*) and sign bit of (S*j*) to S*i*
051*ijk*	S*i* S*j*!S*k*	Logical sum of (S*j*) and (S*k*) to S*i*
051*i*0*k*	S*i* S*k*	Transmit (S*k*) to S*i*
051*ij*0	S*i* S*j*!SB	Logical sum of (S*j*) and sign bit to S*i* (S*j* ≠ 0)
051*i*00	S*i* SB	Enter sign bit into S*i*

Table 5. Address Logical Functional Unit Instructions

Instruction	CAL	Description
044*ijk*	A*i* Aj&A*k*	Logical product of (A*j*) and (A*k*) to A*i*
045*ijk*	A*i* #A*k*&A*j*	Logical product of (A*j*) and one's complement of (A*k*) to A*i*
046*ijk*	A*i* A*j*\A*k*	Logical difference of (A*j*) and (A*k*) to A*i* (A*j* ≠ 0)
047*ijk*	A*i* #A*j*\A*k*	Logical equivalence of (A*k*) and (A*j*) to A*i*
047*i*0*k*	A*i* #A*j*	Transmit one's complement of (A*k*) to A*i*
050*ijk*	A*i* A*j*!A*i*&A*k*	Logical product of (A*i*) and (A*k*) complement ORed with logical product of (A*j*) and (A*k*)
051*ijk*	A*i* A*j*!A*k*	Logical sum of (A*j*) and (A*k*) to A*i*

Address and Scalar Mask

The address mask and scalar mask functions are not scalar logical operations, but are included in this section. Address and scalar mask functions use instructions 042*ijk* and 043*ijk*. Refer to Table 6 and Table 7 for the scalar and address mask instruction formats, respectively.

Table 6. Scalar Mask Instructions

Instruction	CAL	Description
042*ijk*	S*i*<*exp*	Form ones mask in S*i* *exp* bits from the right; *jk* field = 100 – *exp*
042*i*77	S*i* 1	Enter 1 into S*i*
042*i*00	S*i*-1	Enter -1 into S*i*; (S*i* = 177777 177777 177777 177777)
043*ijk*	S*i*>*exp*	Form ones mask in S*i* *exp* bits from the left: *jk* field = *exp*
043*ijk*	S*i* #<*exp*	Form zeroes mask in S*i* *exp* bits from the right: *jk* field gets 100_8= *exp*
043*i*00	S*i* 0	Clear S*i*

Table 7. Address Mask Instructions

Instruction	CAL	Description
042*ijk*	A*i*<*exp*	Form ones mask in A*i* *exp* bits from the right; *jk* field = 100 – *exp*
042*i*77	A*i* 1	Enter 1 into A*i*
042*i*00	A-1	Enter -1 into A*i*; (A*i* = 177777 177777 177777 177777)
043*ijk*	A*i*>*exp*	Form ones mask in A*i* *exp* bits from the left: *jk* field = *exp*
043*ijk*	A*i* #<*exp*	Form zeroes mask in A*i* *exp* bits from the right: *jk* field gets 100_8 = *exp*
043*i*00	A*i* 0	Clear A*i*

The address and scalar mask functional unit is located on the SS options. When the 042*ijk* or 043*ijk* instruction issues, the *jk* field is sent from the BU0 option. The *jk* field determines how many 1 bits are set, and the *h* field bit 0 determines whether the mask should be formed from the left or the right. Figure 12 is a block diagram of the scalar mask functional unit.

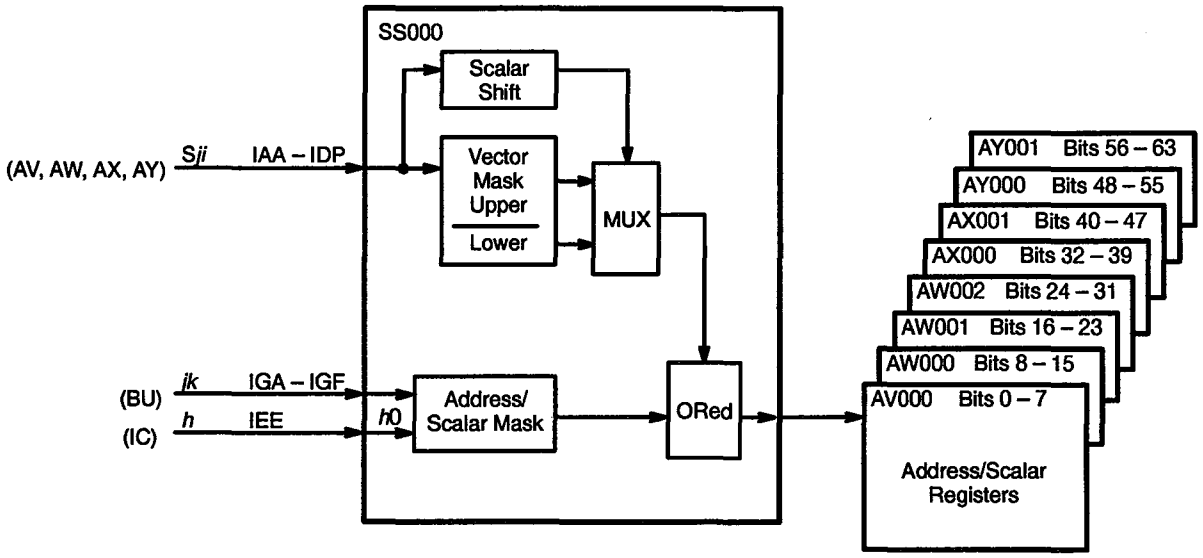


Figure 12. Scalar Mask Block Diagram

Transmit *nm* to S*i*, S*i* Upper, S*i* 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 S*i* Instructions

Instruction	CAL	Description
040*i*00*nm*	S*i exp*	Transmit expression = *nm* to S*i*, bits 0 through 31 (bits 32 through 63 = 0)
040*i*20*nm*	S*i* S*i*:*exp*	Transmit expression = *nm* to S*i*, bits 0 through 31 (bits 32 through 63 unchanged) (*j*2 = 0)
040*i*40*nm*	S*i exp*:S*i*	Transmit expression = *nm* to S*i*, bits 32 through 63 (bits 0 through 31 unchanged) (*j*2 = 1)
041*i*00*nm*	S*i exp*	Transmit expression = one's complement of *nm* to S*i*, bits 0 through 31 (S*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 as designated by the *k* field of instruction 026*ijk* (*k* = 0 or 1 for S registers, and *k* = 2 or 3 for A registers). The maximum count is 100_8 (64_{10}) for the corresponding number of 1 bits set in the A or S register. The smallest count is zero, which occurs 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 the instruction is 026*ij*0/2, all 7 bits of the final population count are sent to the A register. When a 026*ij*1/3 instruction is issued, the entire S or A register is counted for the number of 1 bits set, but 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 the A*i* register. To use the address/scalar leading zero count functional unit, a 027*ij*0 instruction is issued where S*j* is the operand and A*i* is the result register. The 027*ij*1 instruction 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, and the result is 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
026*ij*0	A*i* PS*j*	Transmit population count of (S*j*) to A*i*
026*ij*1	A*i* QS*j*	Transmit population count parity of (S*j*) to A*i*
026*ij*2	A*i* PA*j*	Transmit population count of (A*j*) to A*i*
026*ij*3	A*i* QA*j*	Transmit population count parity of (A*j*) to A*i*
027*ij*0	A*i* ZS*j*	Transmit leading zero count of (S*j*) to A*i*
027*ij*1	A*i* ZA*j*	Transmit leading zero count of (A*j*) to A*i*

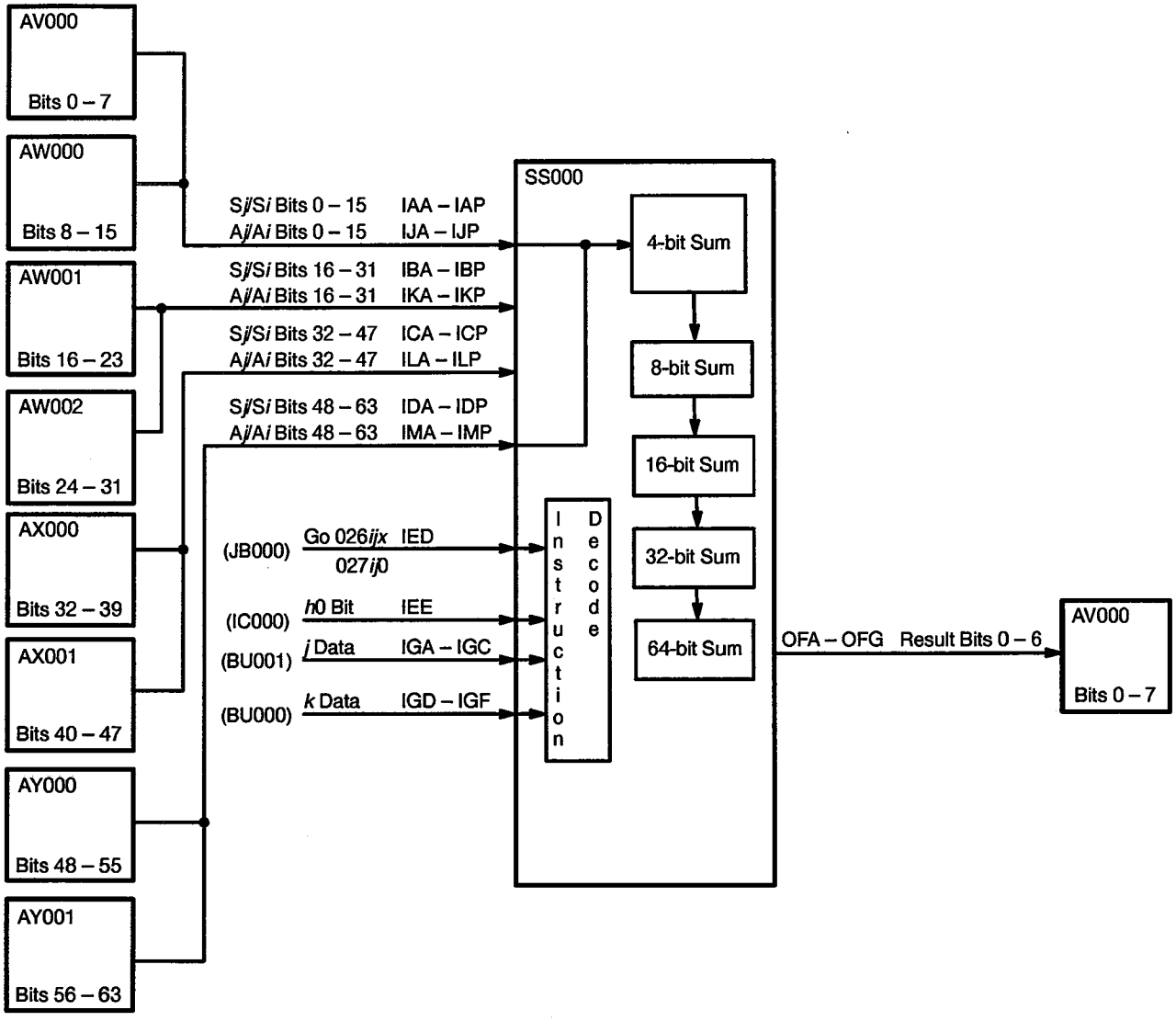


Figure 13. A/S Population/Parity/Leading Zero Count

ADDRESS REGISTER SHIFT

The address register shift function is performed on the SS option (refer to Figure 22 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 shifts (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 completely off the register, leaving the register cleared.

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.

Table 10. Address Register Shift Instructions

Instruction	CAL	Description
052*ijk*	A0 A*i*<*exp*	Shift (A*i*) left *exp* = *jk* places to A0
053*ijk*	A0 A*i*>*exp*	Shift (A*i*) right *exp* = 100_8–*jk* places to A0
054*ijk*	A*i* A*i*<*exp*	Shift (A*i*) left *exp* = *jk* places to A*i*
055*ijk*	A*i* A*i*>*exp*	Shift (A*i*) right *exp* = 100_8–*jk* places to A*i*
056*ijk*	A*i* A*i*, A*j*<A*k*	Shift (A*i*) and (A*j*) left (A*k*) places to A*i*
056*ij*0	A*i* A*i*, A*j*<1	Shift (A*i*) and (A*j*) left one place to A*i*
056*i*0*k*	A*i* A*i*<A*k*	Shift (A*i*) left (A*k*) places to A*i*
057*ijk*	A*i* A*j*, A*i*>A*k*	Shift (A*j*) and (A*i*) right (A*k*) places to A*i*
057*ij*0	A*i* A*j*, A*i*>1	Shift (A*j*) and (A*i*) right one place to A*i*
056*i*0*k*	A*i* A*i*>A*k*	Shift (A*i*) right (A*k*) places to A*i*

Address Register Single Shift

The address register single-shift instructions are 052*ijk* through 055*ijk*. The first two instructions perform left single shifts (052*ijk*) and right single shifts (053*ijk*) on the content of the A*i* register and always store the result in A0. 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 054*ijk* and 055*ijk* 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 similarly 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 056*ijk* (left double shift) and 057*ijk* (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 AV option sends 7 bits of shift count to the SS option. With both single and double shifts, the breakdown of the shift count is nearly the same, except that the double shift has 1 extra bit (bit 6). Refer to Figure 14 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 14. 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: All references to shift counts in this documentation are in decimal notation.

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) and the unit shifts the data left 23_{10} places.

The hardware that performs the shifts is the same for both left and right shifts. In reality, the hardware can perform 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 15 illustrates how a left single shift is performed for a 054220 instruction. (A*i* A*i*<*exp*), shift A2 left *jk* places (20_8) with data bit 10 set.

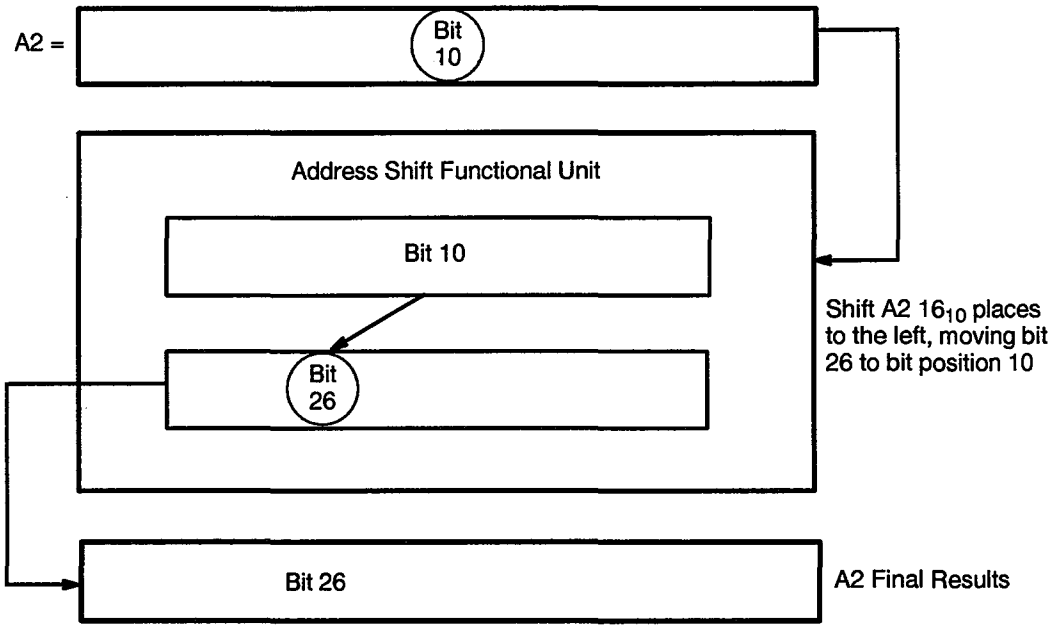


Figure 15. Address Register Left Single Shift

Address Register Right Single Shift

Figure 16 illustrates 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 exp*) that shifts A*i* right *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. The content of A2 is shifted to the left 54_8 places to set bit 25 of the result.

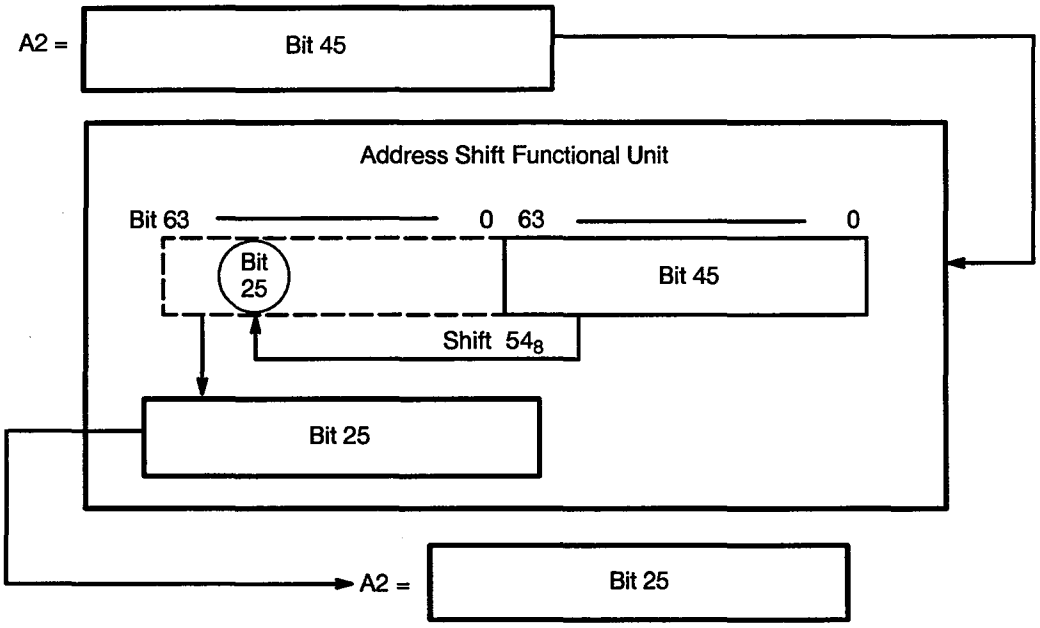


Figure 16. Address Register Right Single Shift

NOTE: On a right shift, the programmer is responsible for converting the shift count to a two's complement value and supplying that value to the functional unit.

Address Register Left Double Shift

Double shift instructions execute in the same manner as single shifts except that the double shift concatenates two 64-bit registers to form a value. Figure 17 illustrates a left double shift using a 056123 instruction (A*i* A1, A*j*<A*k*). In this example, (A*i*) and (A*j*) left shift (A*k*) places to A*i*. A3 = 40_8 (32_{10}), A1 has bit 30 set, and S2 has 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*.

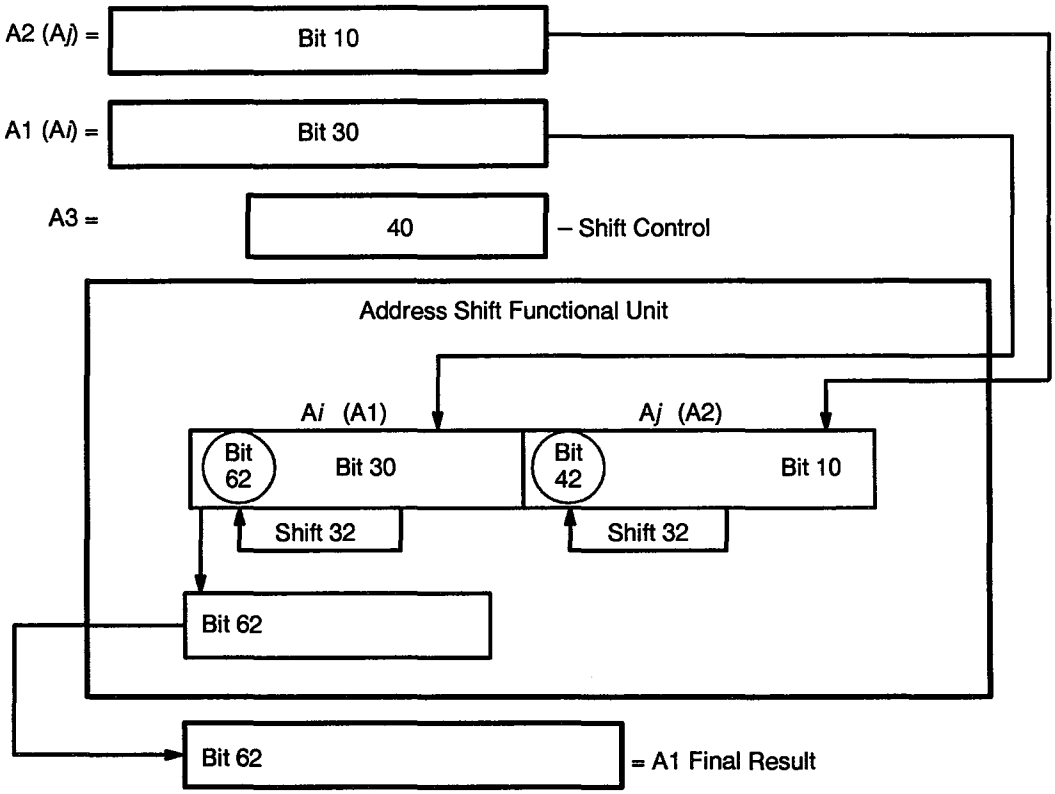


Figure 17. Address Register Left Double Shift

Shifting A*i* and A*j* to the left 32 places puts bit 30 of A1 at bit position 62 and bit 10 of A2 at bit position 41. Because bit 41 of A2 does not transfer to the result register (A1), it is lost. The result bit (bit 62) is sent to the A*i* (A1) register. The A*j* (A2) register remains unchanged.

Address Register Right Double Shift

To perform an address register right double shift, a 057*ijk* [(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 18 illustrates a 057123 instruction with the indicated parameters.

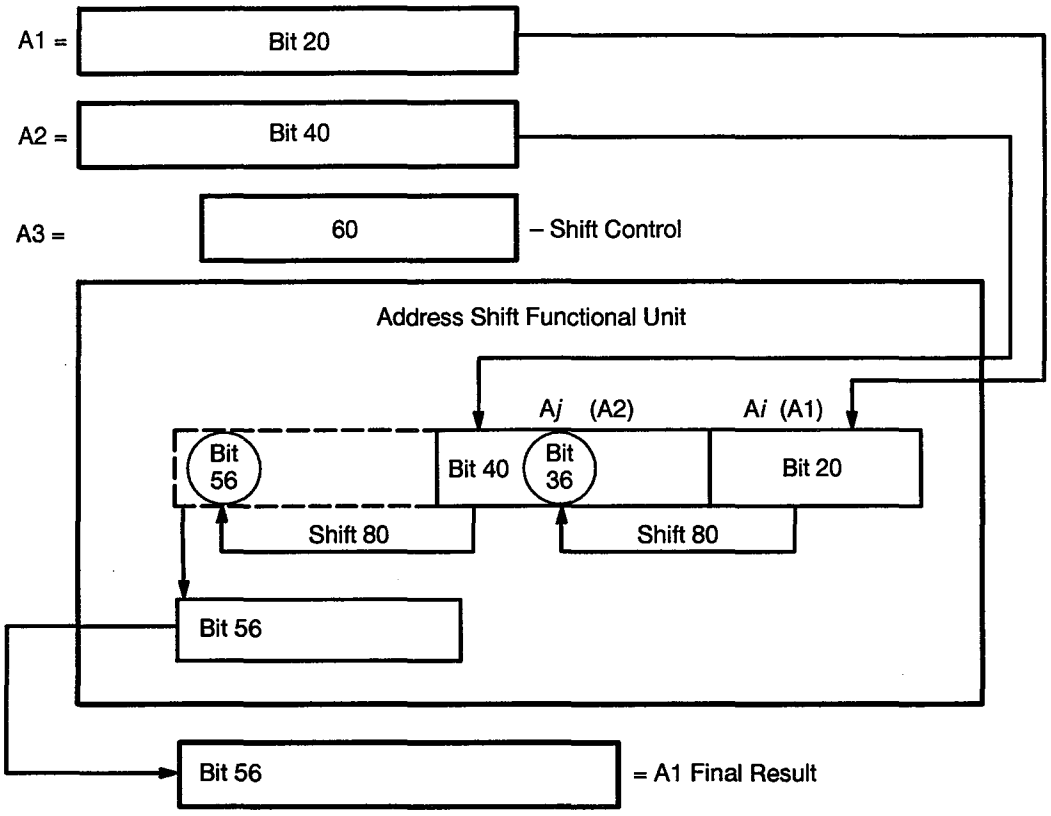


Figure 18. Address Register Right Double Shift

To right shift A*j* and A*i* using left shifts, the two's complement is first performed on A3, 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} moves bit 40 of A2 to bit position 56 inside the dotted box and bit 20 of A1 to bit position 36 of A2. Because bit 36 does not transfer into the result register (indicated by the dotted box), it is lost. Bit 56 is sent to the final result register (A1).

Left Single-shift Instruction

Refer to Figure 19 when reading the following two examples of the address register left single-shift instruction.

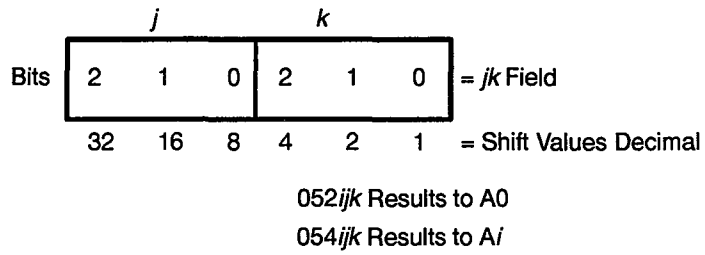


Figure 19. Example of an A Register Left Single-shift Instruction

Example 1: Write the instruction to shift the contents of A2 left 20_{10} places and put the result into A0.

Steps:
1. 052*ijk* – left shift instruction result goes to A0
2. *jk* field – shift count $20_{10} = 24_8 = jk$ field
3. 052224 – final instruction

Example 2: Write the instruction to shift A4 left 35_{10} places and put the result into A4.

Steps:
1. 054*ijk* – left shift instruction result goes to A*i*
2. *jk* field – shift count $35_{10} = 43_8$
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.

- 053*ijk* results to A0
- 055*ijk* results to A*i*

Example 1: Write the instruction to shift A5 right 10_{10} places and put the result into A0.

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

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

two's complement = 110101

$$\underline{+\ 1}$$

$$110110 = 66_8$$

3. 053566 – final instruction

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

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

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

two's complement = 100011

$$\underline{+\ 1}$$

$$100100 = 44_8$$

or $100_8 - 34_8 = 44_8$

3. 055744 – final instruction

Left Double-shift Instruction

Refer to Figure 20 when reading the following example of an address register left double-shift instruction.

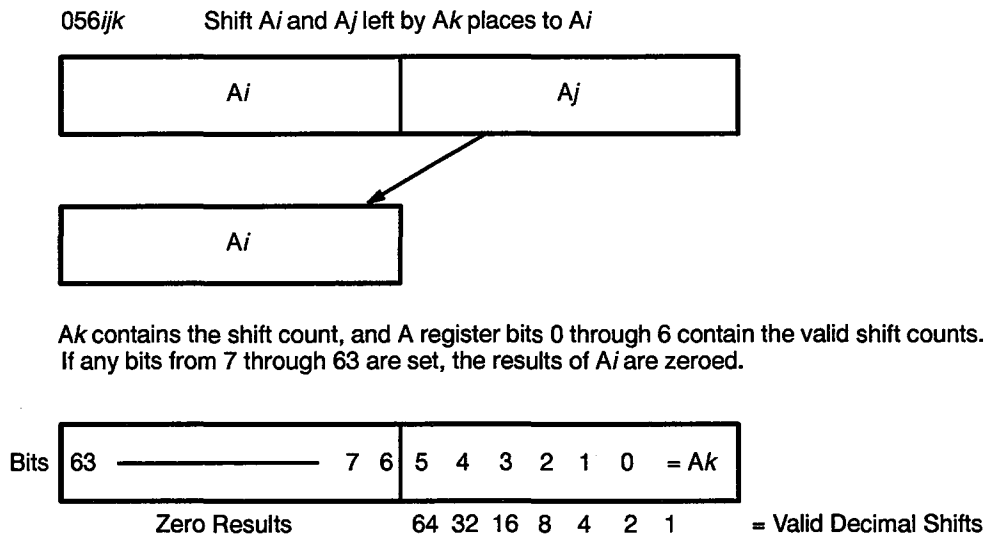


Figure 20. Example of an Address Register Left Double-shift Instruction

Example 1: Write the instruction to double shift A2 and A3 left 64_{10} places and put the results into A2.

056234 – final instruction, where A4 – 100_8

NOTE: A 056 instruction with $i = j$ and $(Ak) \leq 64$ effects a circular left shift.

Right Double-shift Instruction

Refer to Figure 21 when reading the following example of a scalar right double-shift instruction.

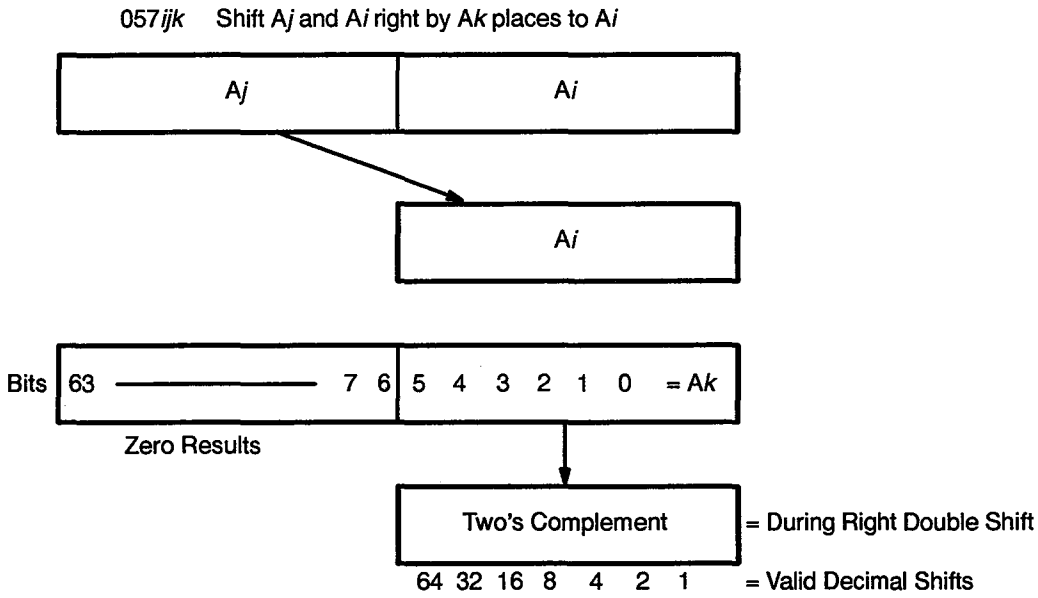


Figure 21. Example of an Address Register Right Double-shift Instruction

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 occur inside the address shift functional unit.

Example 1: Write the instruction to double shift right A4 and A5 32_{10} places and put the result into A4.

057454 – final instruction, where A4 = 40_8 hardware generates a shift count of 140_8 inside the functional unit.

NOTE: Issue a 057 instruction with $i = j$ and (A*k*)$\leq$ 64 to effect a circular right shift.

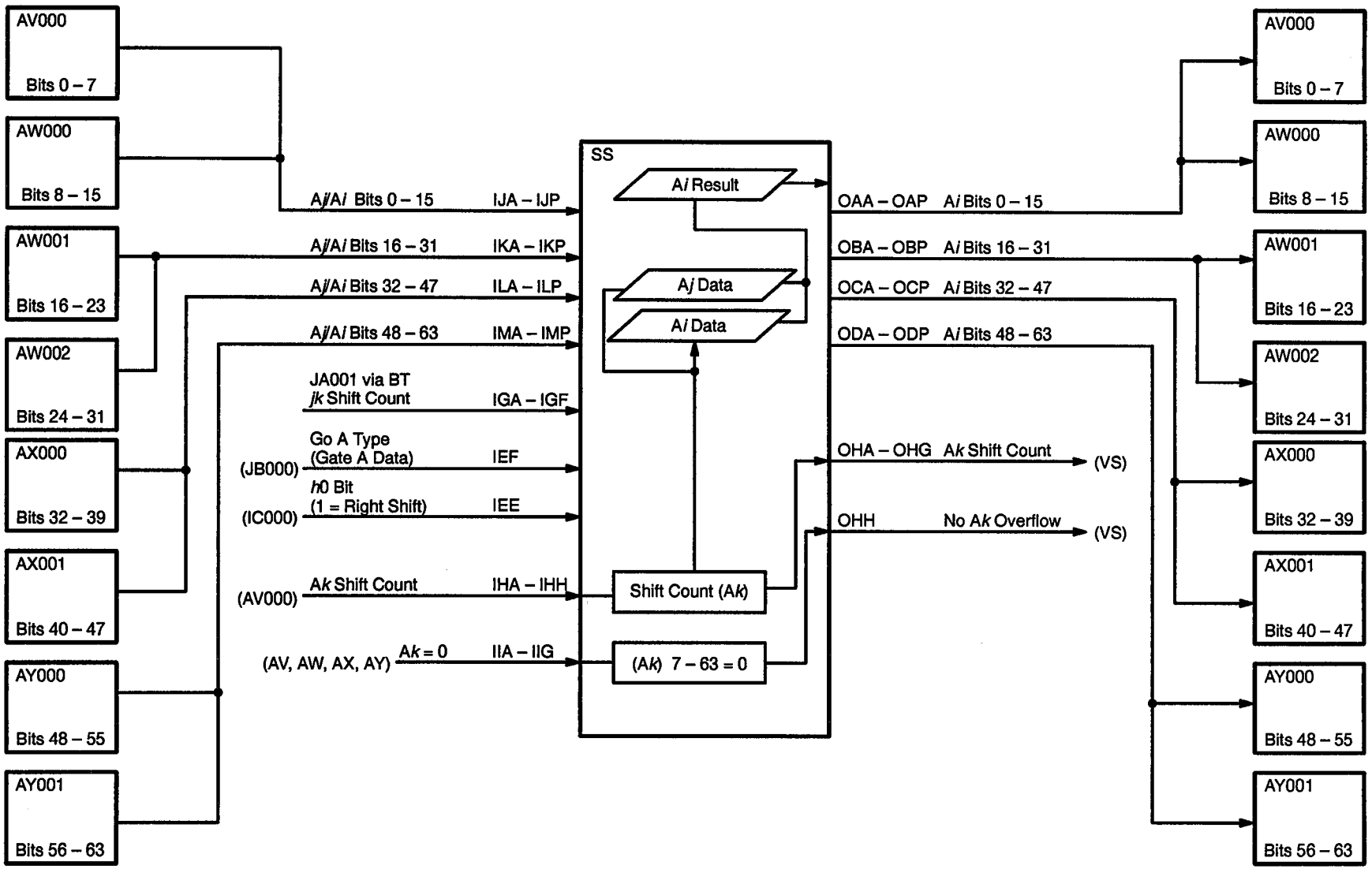


Figure 22. Address Register Shift

SCALAR SHIFT

The scalar shift function is performed on the SS option (refer to Figure 31 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 shifts (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
052*ijk*	S0 S*i*<*exp*	Shift (S*i*) left *exp* = *jk* places to S0
053*ijk*	S0 S*i*>*exp*	Shift (S*i*) right *exp* = 100_8 – *jk* places to S0
054*ijk*	S*i* S*i*<*exp*	Shift (S*i*) left *exp* = *jk* places to S*i*
055*ijk*	S*i* S*i*>*exp*	Shift (S*i*) right *exp* = 100_8 – *jk* places to S*i*
056*ijk*	S1 S*i*, S*j*<A*k*	Shift (S*i*) and (S*j*) left (A*k*) places to S*i*
056*ij*0 †	S1 S*i*, S*j*<1	Shift (S*i*) and (S*j*) left 1 place to S*i*
056*i*0*k* ‡	S1 S*i*<A*k*	Shift (S*i*) left (A*k*) places to S*i*
057*ijk*	S*i* S*j*, S*i*>A*k*	Shift (S*j*) and (S*i*) right (A*k*) places to S*i*
057*ij*0 †	S1 S*j*, S*i*>1	Shift (S*j*) and (S*i*) right 1 place to S*i*
057*i*0*k* ‡	S1 S*i*>A*k*	Shift (S*i*) right (A*k*) places to S*i*

† If *j* = 0, then (S*j*) = 0.

‡ If *k* = 0, then (A*k*) = 1.

Scalar Single Shift

The scalar single-shift instructions are 052*ijk* through 055*ijk*. The first two instructions perform single shifts left (052*ijk*) and right (053*ijk*) on the contents of the S*i* register and always store the result in S0. The shift count is obtained from the *jk* field of the instruction. How the value is

represented in the *jk* field for single-shift instructions depends on whether the shift is left or right. For a single left shift, the value in the *jk* field represents the number of octal places (in the range of 0 to 77_8 places) to shift S*i*. For a right shift, the *jk* field is equal to the two's complement of the actual number of places to shift right. If a shift of 24_8 places were required, 54 would be entered in the *jk* field (the two's complement of 24 is 54).

When instructions are written in machine code, the programmer is responsible for complementing the shift count. However when instructions are written in CAL, the assembler performs this operation automatically; that is, in the CAL instruction, simply enter the shift count in the range of 1 to 100_8 places. Because the two's complement of the shift count is used for a single shift, a right shift of 0 places is not possible.

The 054*ijk* and 055*ijk* 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 are similar to single shifts; all shifts are end-off with zero fill. However, 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 056*ijk* (double left shift) and 057*ijk* (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.

A double shift uses a 128-bit operand and shifts are end-off with zero fill. Therefore a shift equal to or greater than 128_{10} (200_8) produces a result of zero. The shift count is determined by bits 0 through 6 of the A*k* register, providing a shift range 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 AV000 option sends the shift count to the SS option. All eight A-series options check the value of the 64-bit A register to determine if any bits greater than bit 6 have been set. If any of these bits are set, the

result is lost due to overshift. If each A-series option reports that its bits are zero, the shift count is valid and a signal called Ak = 0 is sent to the SS option.

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 similar, except that the double shift has 1 extra bit (bit 6). Refer to Figure 23 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 23. 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 this document 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 are 16, 4, 2, and 1, respectively. The sum of the shift values is 23 (16 + 4 + 2 + 1), and the instruction shifts the data left 23_{10} places.

The hardware that performs the shifts is the same for both left and right shifts. (Actually, the physical hardware can perform only left shifts.) Right shifts are achieved by the way in which the data is entered into the shifter and by the use of two's complement values for shift counts.

Scalar Left Single Shift

Figure 24 is an illustration of how a left single shift is performed for a 054220 instruction (*Si Si<exp*). In the following example, the contents of S2 (data bit 10 set) are shifted left 20_8 places (16_{10}), and the result is returned to S2.

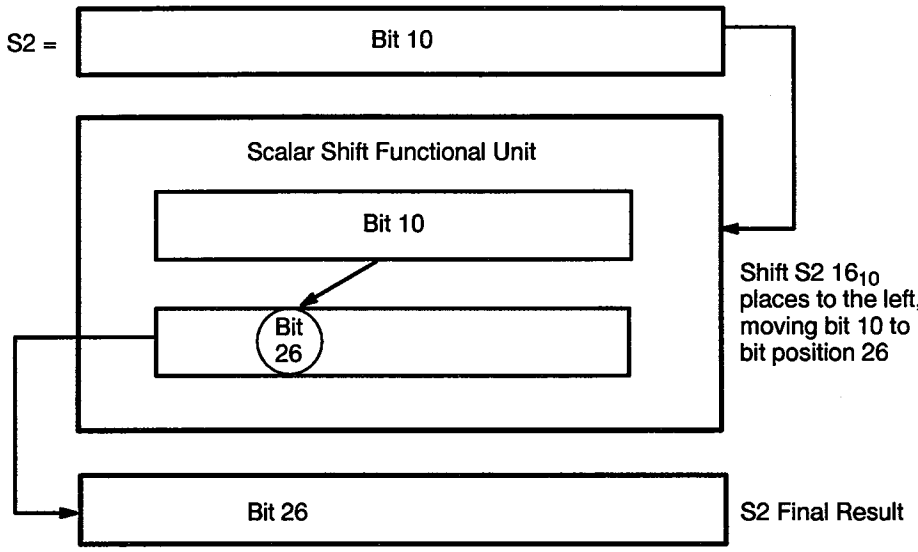


Figure 24. Scalar Left Single Shift

Scalar Right Single Shift

Figure 25 illustrates 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 exp*) that shifts S*i* right exp = 100 – *jk* places to S*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 S2 to be shifted to the left 54_8 places to set bit 25 of the result.

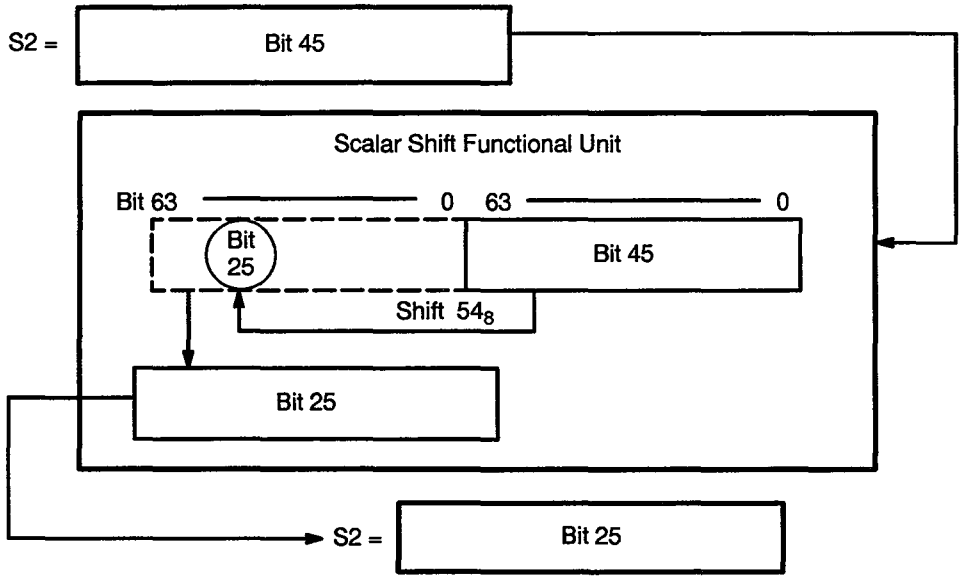


Figure 25. Scalar Right Single Shift

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 similar to single shifts except that they concatenate two 64-bit registers to form a value. Figure 26 illustrates a left double shift using a 056123 instruction (S*i*, S*j* < A*k*). In this example, S (S*i*) and (S*j*) shift left (A*k*) places to S*i*. A*k* = A3 = 40_8 (32_{10}). Initially, bit 30 is set in S1, and bit 10 is set in S2. During a left double shift, the content of S*j* moves into S*i*. The two registers are concatenated as illustrated, with S*i* ahead of S*j*.

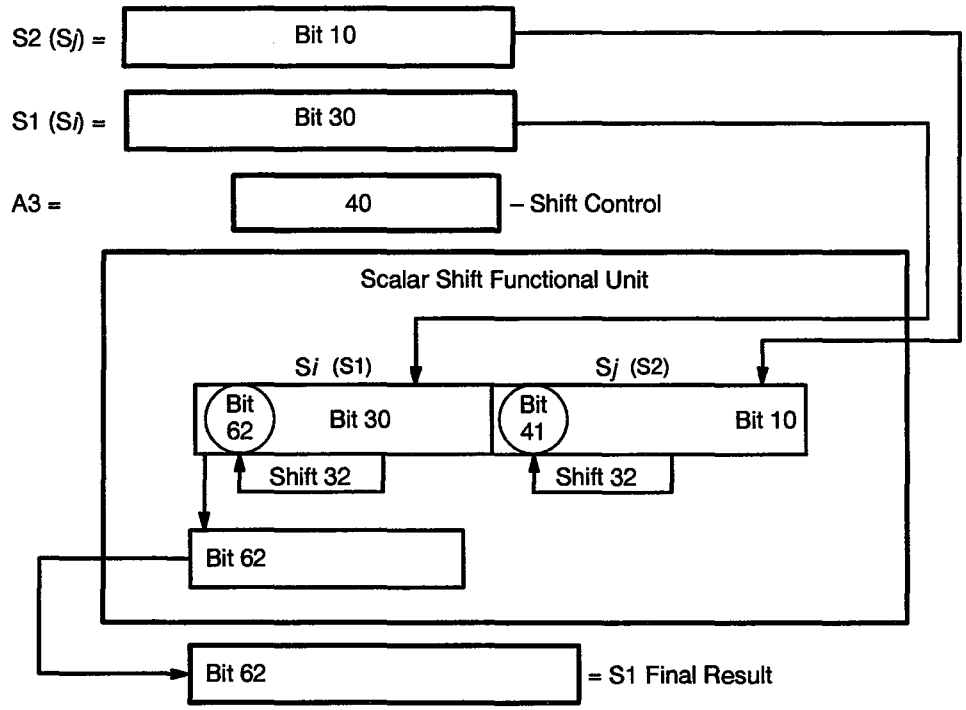


Figure 26. Scalar Left Double Shift

Shifting S*i* and S*j* to the left 32 places puts bit 30 of S1 at bit position 62 and bit 10 of S2 at bit position 41. Bit 41 of S2 does not enter the result register S1 and is lost. The result bit (bit 62) is then sent to the S*i* (S1) register. The content of register S*j* (S2) remains unchanged.

Scalar Right Double Shift

A 057*ijk* instruction (S*i* S*j*, S*i* > A*k*) shifts (S*j*) and (S*i*) right (A*k*) places to S*i*. Figure 27 illustrates a 057123 instruction with the indicated parameters.

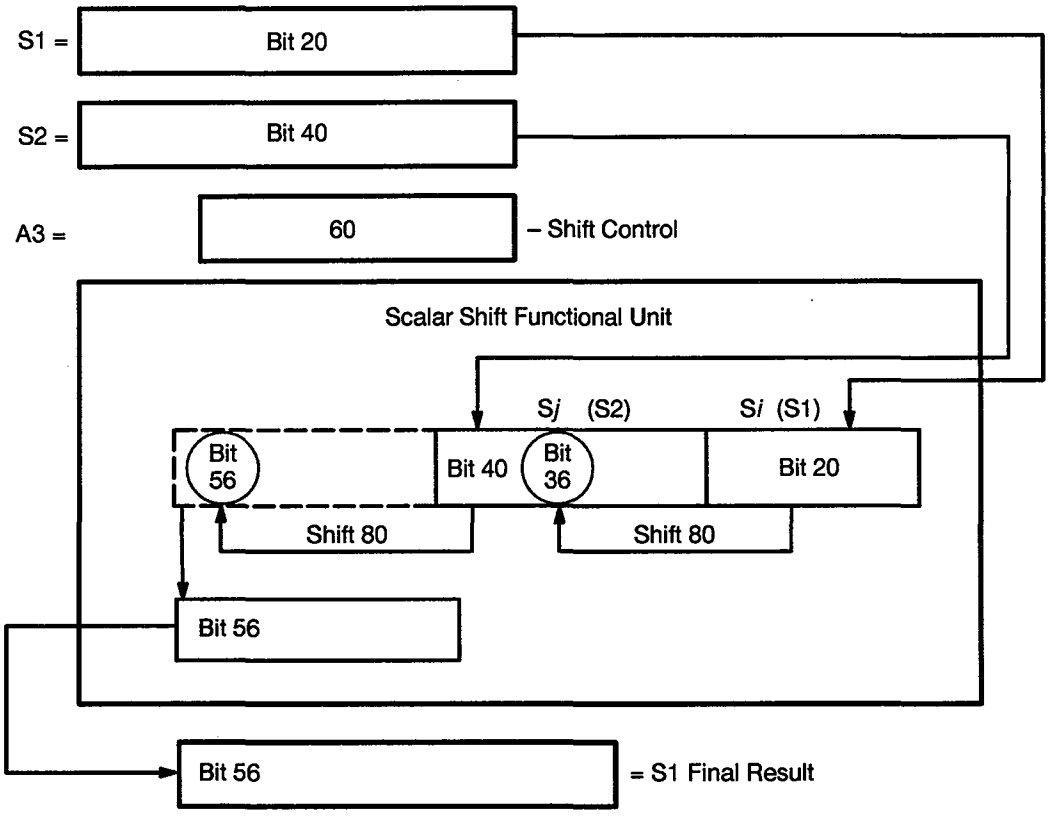


Figure 27. Scalar Right Double Shift

To right shift S*j* and S*i* using left shift operations, the content of A3, which currently equals 60_8 (48_{10}) is converted into a two's complemented value. The two's complement of 60_8 is 120_8 (or 1010000_2 or 80_{10}). The required shift can be accomplished through successive shifts of 64_{10} and 16_{10}. A left shift of 80_{10} moves bit 40 in S2 to bit position 56 inside the dotted box and bit 20 of S1 to bit position 36 of S2. Because bit 36 does not enter the intermediate result register (indicated by the dotted box), it is lost, and bit 56 is sent to the final result register (S1).

Left Single-shift Instruction

Refer to Figure 28 while reading the following two examples of the scalar left single-shift instruction:

- 052*ijk*, results to S0
- 054*ijk*, results to S1

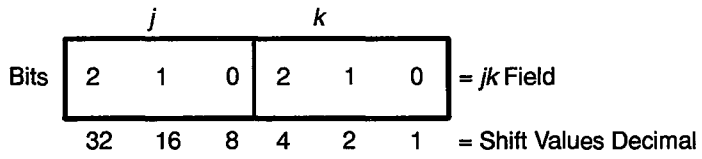


Figure 28. Example of a Scalar Left Single-shift Instruction

Example 1: Write the instruction that shifts S2 left 20_{10} places, and places the results into S0.

Steps:
1. 052*ijk* – left shift instruction result goes to S0
2. *jk* field– shift count $20_{10} = 24_8 = jk$ field
3. 052224 – final instruction

Example 2: Write the instruction that shifts S4 left 35_{10} places, and places the results into S4.

Steps:
1. 054*ijk* – left shift instruction result goes to S*i*
2. *jk* field– shift count $35_{10} = 43_8$
3. 054443 – final instruction

Right Single-shift Instruction

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

- 053*ijk*, results to S0
- 055*ijk*, results to S*i*

Example 1: Write the instruction that shifts S5 right 10_{10} places, and places the results into S0.

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

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

one's complement = 110101
$$\underline{+\ 1}$$

two's complement = $110110 = 66_8$

3. 053566 – final instruction

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

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

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

one's complement = 100011
$$\underline{+\ 1}$$

two's complement = $100100 = 44_8$

or $100_8 - 34_8 = 44_8$

3. 055744 – final instruction

Left Double-shift Instruction

Refer to Figure 29 while reading the following example of a scalar left double-shift instruction: 056*ijk*, Shift S*i* and S*j* left A*k* places to S*i*.

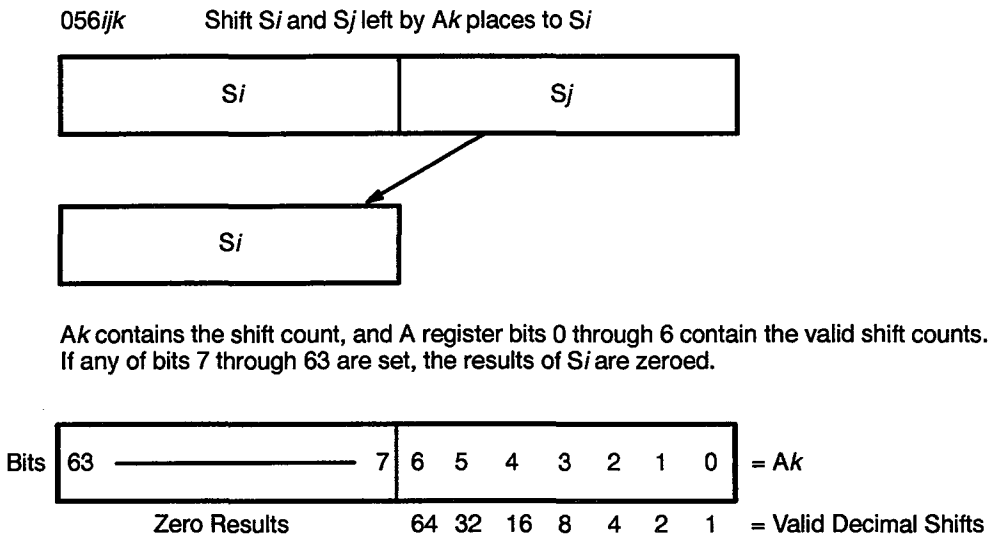


Figure 29. Example of a Scalar Register Left Double-shift Instruction

Example 1: Write the instruction that left double shifts S2 and S3 64_{10} places, and places the result into S2.

Step 1. 056234 – final instruction, where (A4) = 100_8

NOTE: A circular left shift can be achieved by issuing a 056 instruction with $i = j$ and (Ak) ≤ 64.

Right Double-shift Instruction

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

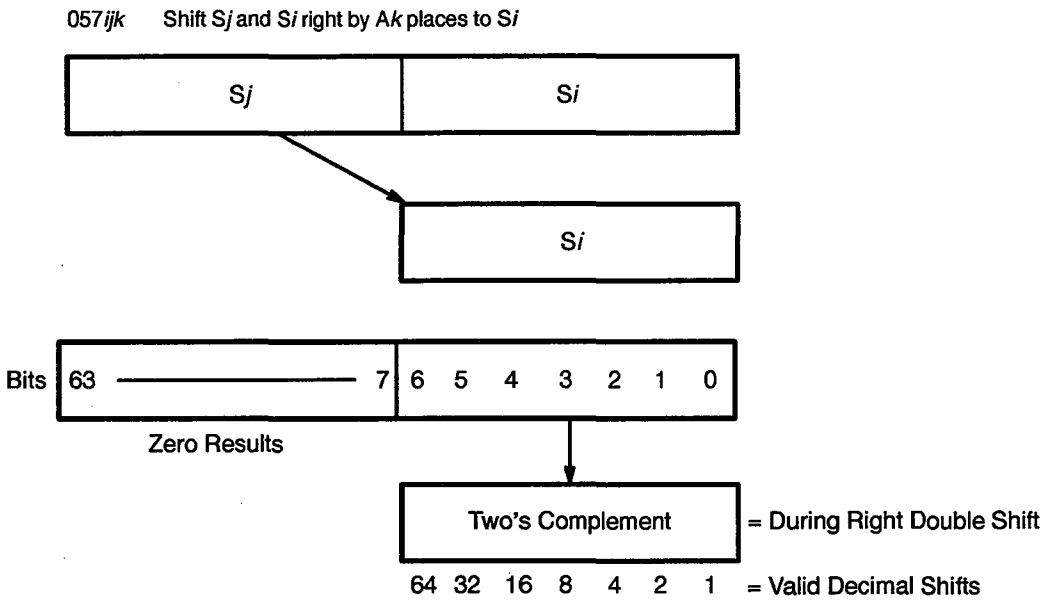


Figure 30. Example of a Scalar Register Right Double-shift Instruction

A*k* contains the shift count, and address (A) register bits 0 through 7 contain the valid shift counts. If any bit in the range from bit 7 through bit 63 is set, the result from S*i* is zeroed. Also, the hardware generates the two's complement of the shift count on the A*k* register bits 0 through 7 for a right double shift.

During 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 occur inside the scalar shift functional unit.

Example 1: Write an instruction to right double shift S4 and S5 32_{10} places, and place the result into S4.

057454 – final instruction, where (A4) = 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.

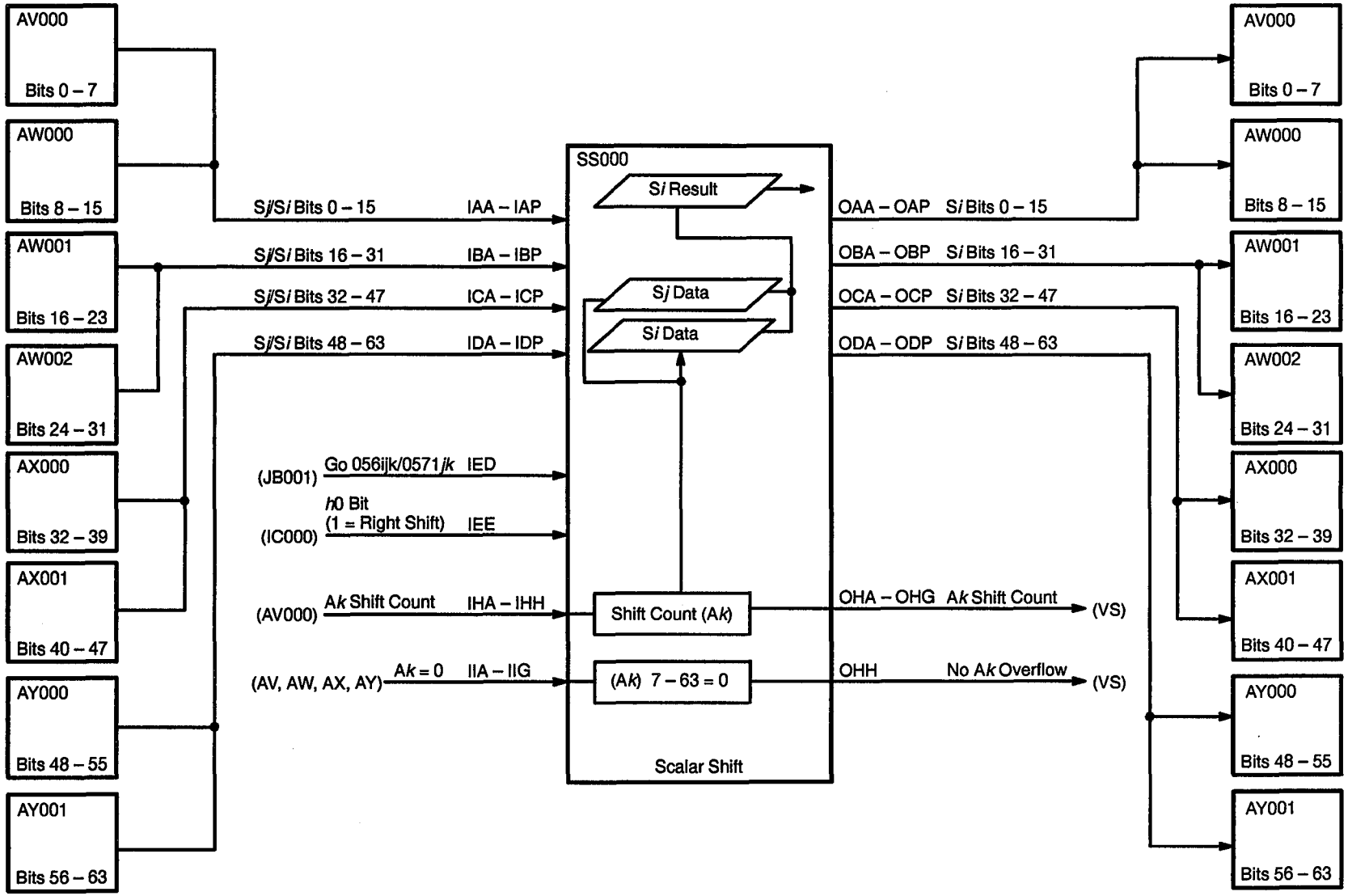


Figure 31. Scalar Shift

ADDRESS MULTIPLY

The AN option performs the address multiply operation (a 032*ijk* instruction). The AN option also distributes (fans out) the A*j* and A*k* operands used for other A register operations.

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 result will fit within 48 bits. If the result exceeds 48 bits, the 64-bit incompatibility signal sets, which sets the Address Multiply Interrupt (AMI) flag 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 of the recode groups form as soon as the data arrives at the AN option (namely, 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 enables a multiply operation to execute on about one-fourth of the logic used in a standard pyramid multiply. Because this method holds the A*k* 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.

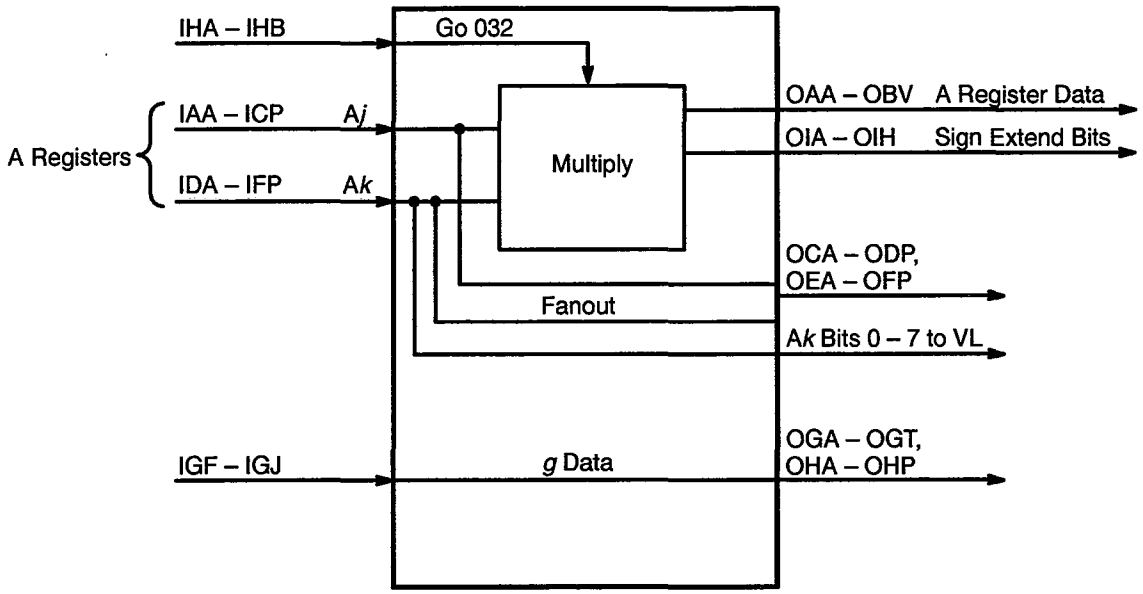


Figure 32. AN Option

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. 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	X47 – X0
0	1	0	+1	X47 – X0
0	1	1	+2	2(X47 – X0)
1	0	0	–2	{2(X47 – X0}'+1
1	0	1	–1	(X47 – X0)'+1
1	1	0	–1	(X47 – X0)'+1
1	1	1	–0	0
i – 1 = Bit to right of recode group		X47 – X0 = Multiplicand		

Standard Binary Multiplication

Refer to the following example of standard binary multiplication.

```
      000011  (3)
      011101  (35)
    ---------
      000011
     000000
    000011
   000011
  000011
000000
---------------
 0000001010111
```

Booth Recode Multiplication

Refer to the following example of Booth Recode multiplication.

```
          000011  (3)
          011101  (35)
        ----------
    000000000011
     11111111010
    00000110
  ---------------
1 000001010111
```

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.

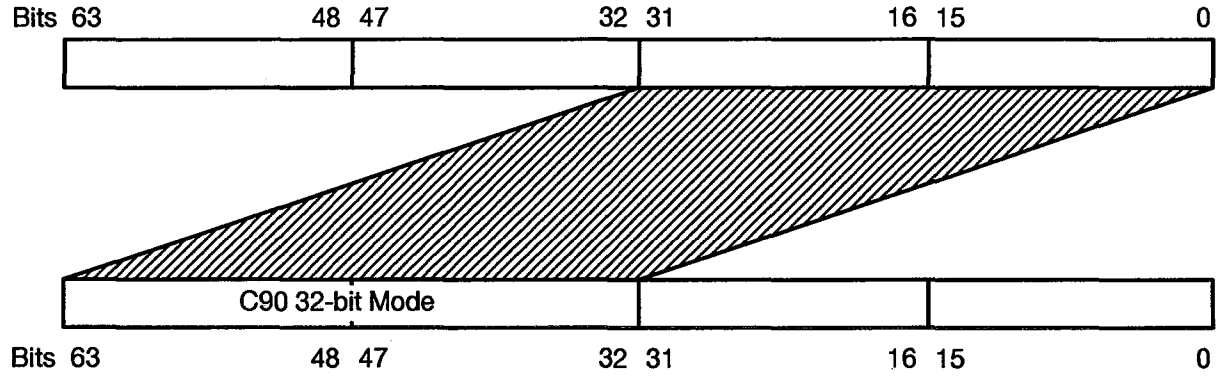
As shown in Table 12, the first recode of the multiplier, bits 1 and 0 and the forced zero, produces a recode value of 010, or +1. The multiplicand is brought down to form the first partial product.

The second recode, bits 3, and 2, and 1, produces a recode value of –1. In this case, the multiplicand is two's complemented and left shifted 1 place.

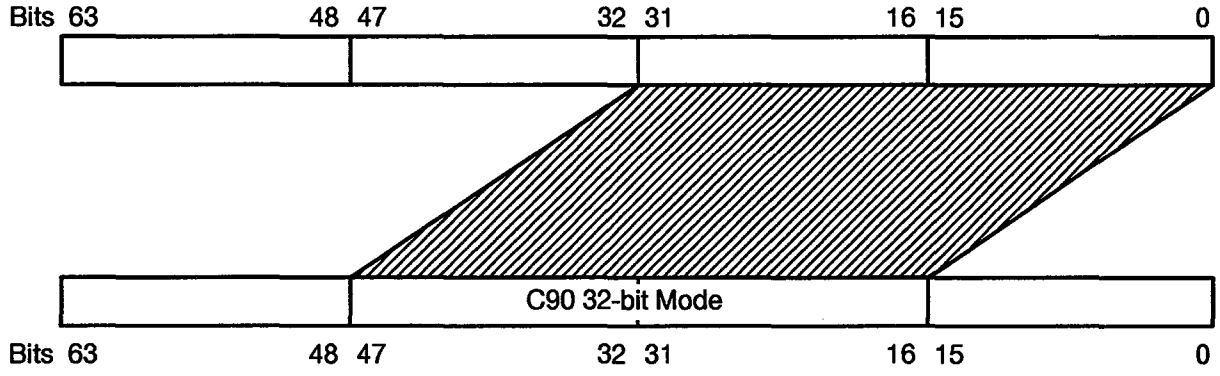
The final recode, bits 5, 4, and 3 produces a recode value of +2. The multiplicand is left shifted 1 place.

INTEGER MULTIPLY

The AM option performs the scalar vector integer multiply operation (166*ijk*). In Triton mode, the AA option receives S*j* and V*k* operands and sends a 40-bit output to V*i* for VL length. In C90 mode, the AA option produces a 32-bit result. To produce the 32 bit result, the S*j* operand must be left shifted 31_{10} places, and the V*k* operand must be left shifted by 16_{10} places before executing the 166*ijk* instruction. (Refer to Figure 33.)



S*j* bits 0 through 31 are gated into bit positions 32 through 63 for Triton mode.



V*k* bits 0 through 31 are gated into bit positions 15 through 47 for C90 mode.

Figure 33. C90 Integer Multiply Mode

The AM option, like the AN option (refer to the "Address Multiply" section), also uses the Booth Recode algorithm for the multiply operation. The AN option performs a leading zero count on the operands to determine whether the results will fit within 40 bit positions. The input operands pass through the floating-point multiply unit before they arrive at the AM option, as shown in Figure 34.

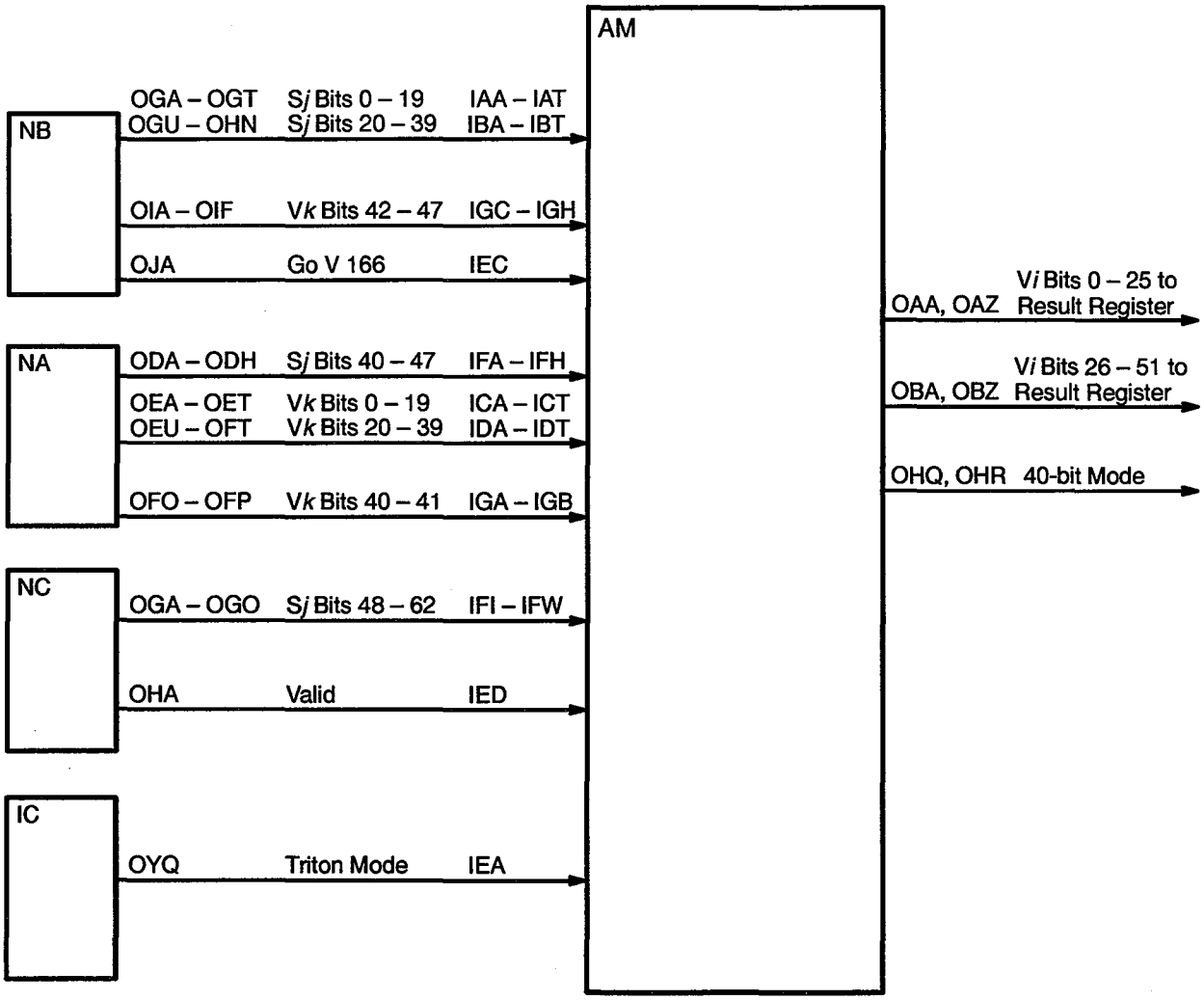


Figure 34. AM Option Inputs

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. These floating-point functional units include floating-point add, floating-point multiply, floating-point divide/square root 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 contained in 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 076*ijk* and 077*ijk* 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. Usually, the *k* field of the instruction selects the vector operand registers, V0 through V7. The *j* field of the instruction indicates either S*j* or V*j*, depending on the instruction. The *i* field of the instruction points to the destination or result register.

When preceded by a 005400 instruction, some vector instructions execute differently. For example, an instruction sequence of 005400 150*ij*0 issues, a left shift of V*j* V0 places to V*i* is performed. Without the preceding 005400 instruction, a 150*ij*0 instruction performs a left shift of V*j* A0 places to V*i*.

The vector registers in the CRAY T90 series system contain a dual set of functional unit pipes. Each functional unit has an identical twin 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
VB	2	Provide read/write address and control (VB0 pipe 0) (VB1 pipe 1) Vector length register Functional unit release
VE	4	Pipe control (VE0,VE1 for pipe 0) (VE2,VE3 for pipe 1)
VN	16	Data multiplexing (VN0 – VN7 pipe 0) (VN8 – VN15 pipe 1) Vector add functional unit Vector logical functional unit
VQ	16	Data multiplexing and storage (VQ0 – VQ7 pipe 0) (VQ8 – VQ15 pipe 1)

VB Option

The two VB options on a CPU module provide vector read and write control. VB0 provides address and control for the even elements of the vector registers, and VB1 provides the address and control for the odd elements. Both VB options have the following common functions:

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

Each VB option also has the following unique features:

- VB0
  - Release vectors for write operations
  - Functional unit release for:
    Vector logical #1
    Vector shift
    Vector floating-point multiply
    Vector divide
  - Even-element addressing
- VB1
  - 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 VB option. There are two VB options, one for each pipe. Both vector length registers are loaded with A*k* data bits 00 through 06 from the AV000 option. These bits are needed to form values from 0 to 177_8. If a value of all 0's is entered, the VL register is forced to a value of 200_8.

A vector length value enters a countdown (decrementing) register. VL bit 0 is removed (pseudo right shifted so that a VL value of 200 becomes a value of 100 in the active register) because each pipe can handle only 100 elements. Every time VL decrements, it generates the **Advance Address** signal. The VB option also checks VL bit 0 to determine whether the vector length is odd or even in order to enable 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 register and the next instruction tries 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 VQ options. If another instruction is waiting for these results or is addressing the same element, the VQ option passes the results directly to the read-out register. The VB option controls 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.

VE Option

There are four VE options on the CP module. VE0 and VE1 control fanout for pipe 0; VE2 and VE3 control fanout for pipe 1. The VE options perform the following functions.

- Instruction parcel data fanout to VQ 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

VN Option

The VN options perform write data multiplexing on an 8-bit slice of all functional unit data. There are 16 VN options. VN000 to VN007 are for even-element steering, and VN008 to VN015 are for odd-element steering.

The VN option performs the following functions:

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

VQ Option

Sixteen VN and VQ options reside on the CP module as illustrated in Table 14. Each option performs read data steering and vector data storage. The read data steering is done on 4-bit slices. The contents of the selected vector register are gated to one of the following destinations:

- 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 VN and VQ 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. VQ000 through VQ007 store vector data for the even elements (pipe 0), and VQ008 through VQ015 store data for the odd elements (pipe 1).

NOTE: VN/VQ options 12 through 15 do not handle exchange data.

Table 14. VN/VQ Data Steering

Option Pipe 0/Pipe 1	VN3/11	VQ3/11	VN2/10	VQ2/10	VN1/9	VQ1/9	VN0/8	VQ0/8
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	VN7/15	VQ7/15	VN6/14	VQ6/14	VN5/13	VQ5/13	VN4/12	VQ4/12
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 VQ 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)
- 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 VN options receive the *i* instruction field from the VE 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.

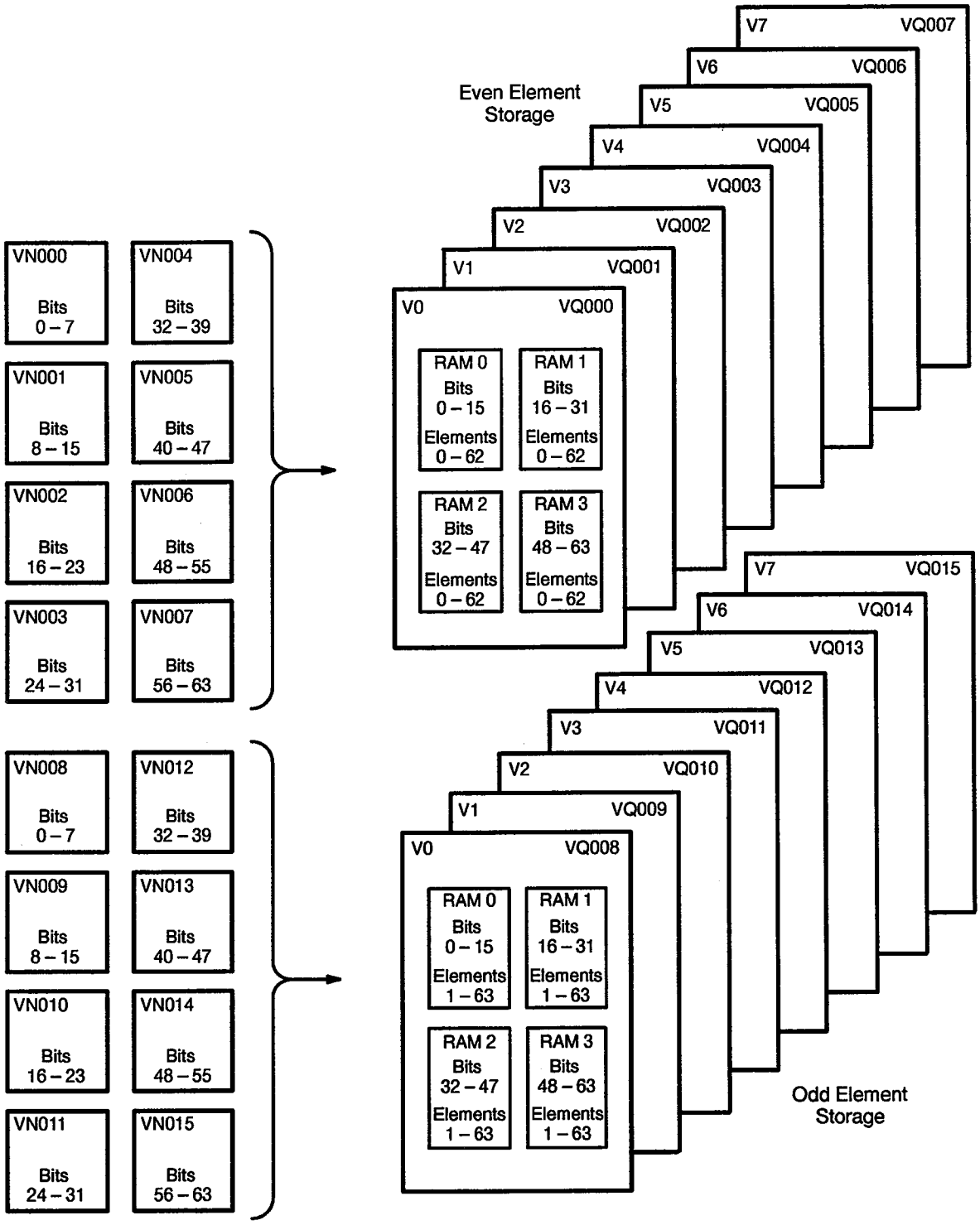


Figure 35. Write Data Path

Read Data Steering

The VN and the VQ options are responsible for read data steering. Each VN and VQ 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 VN and VQ options receive the *j* and *k* fields of the instruction from the VE 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). Also refer 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)

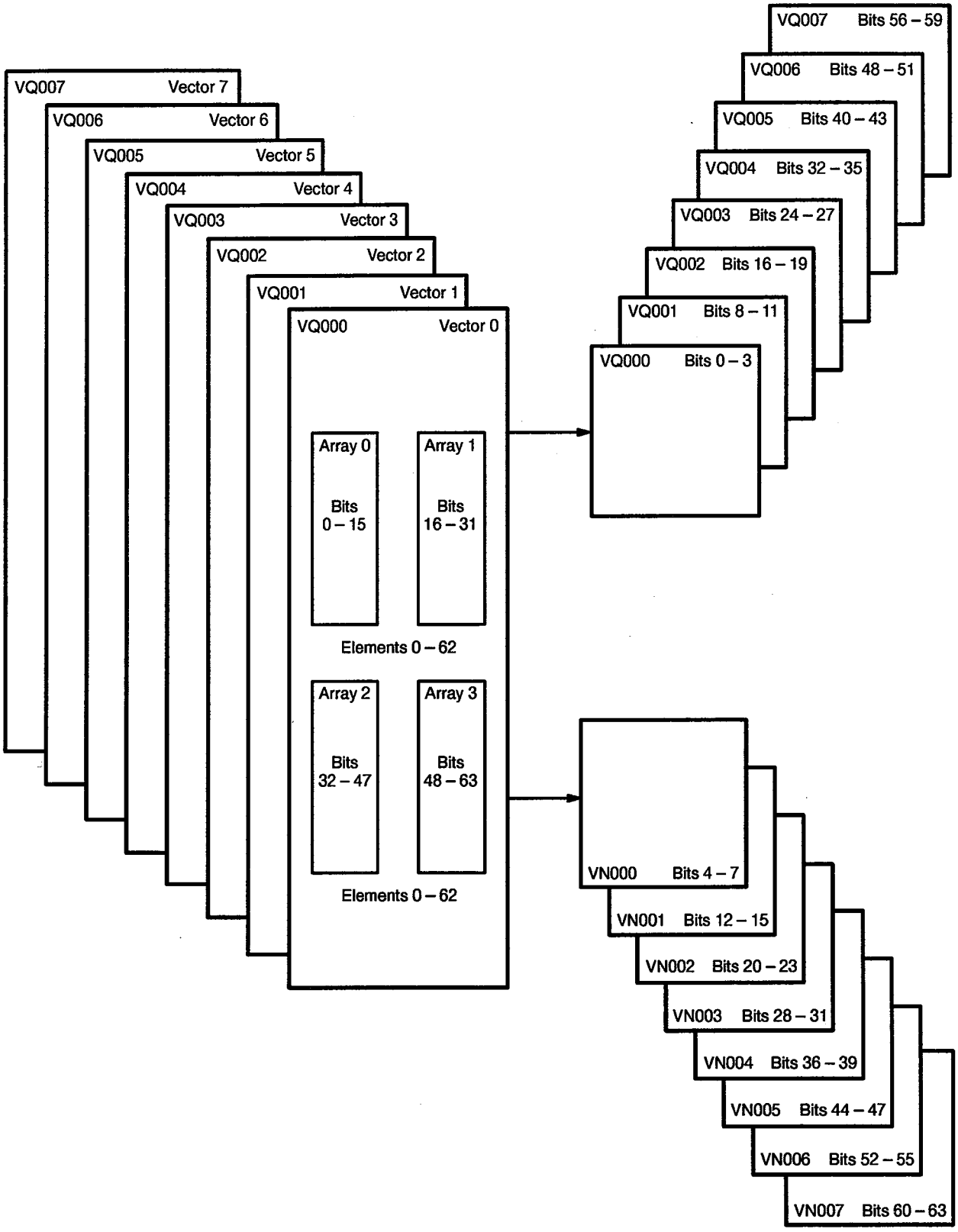


Figure 36. Read Data Path for Pipe 0, Even Elements

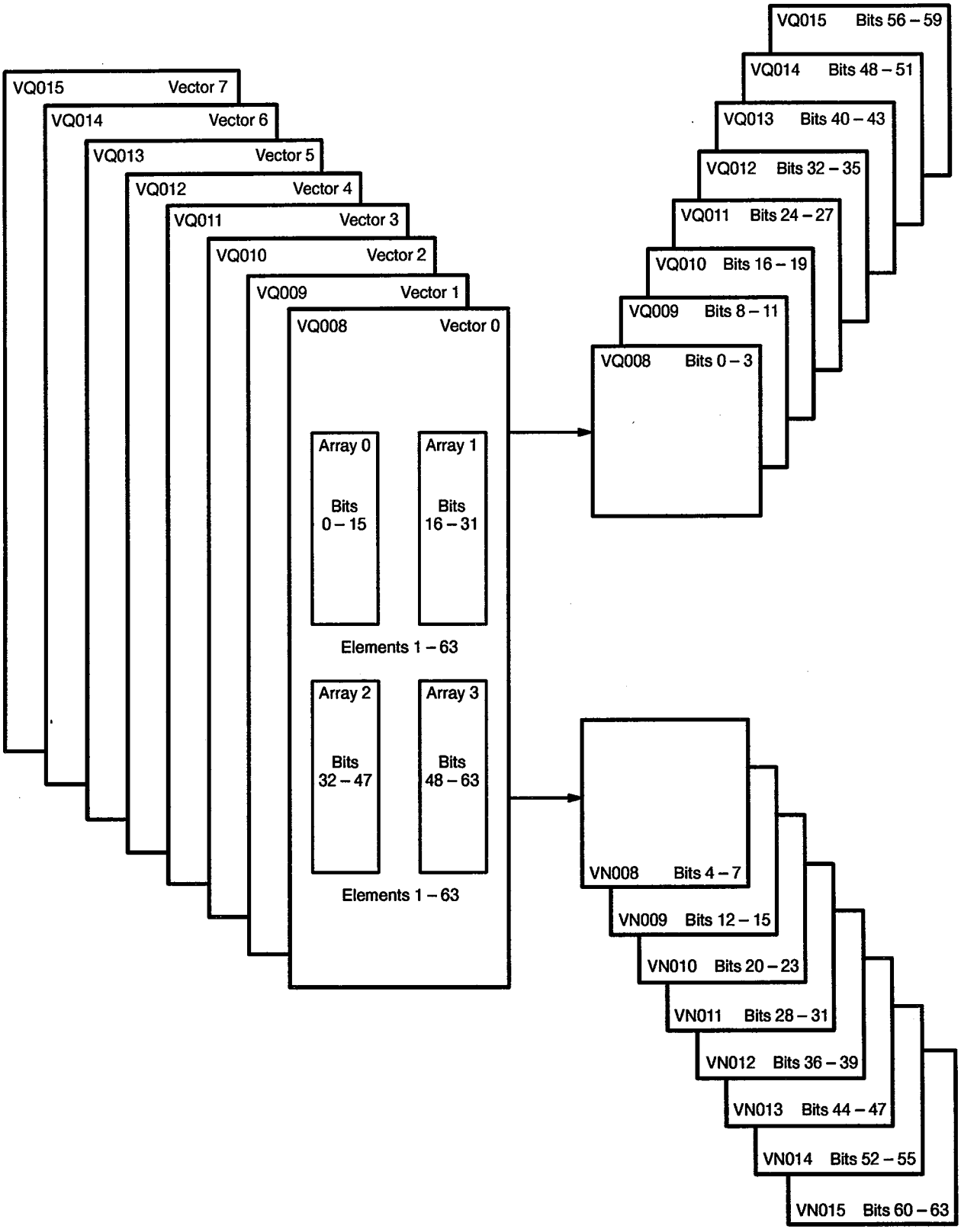


Figure 37. Read Data Path for Pipe 1, Odd Elements

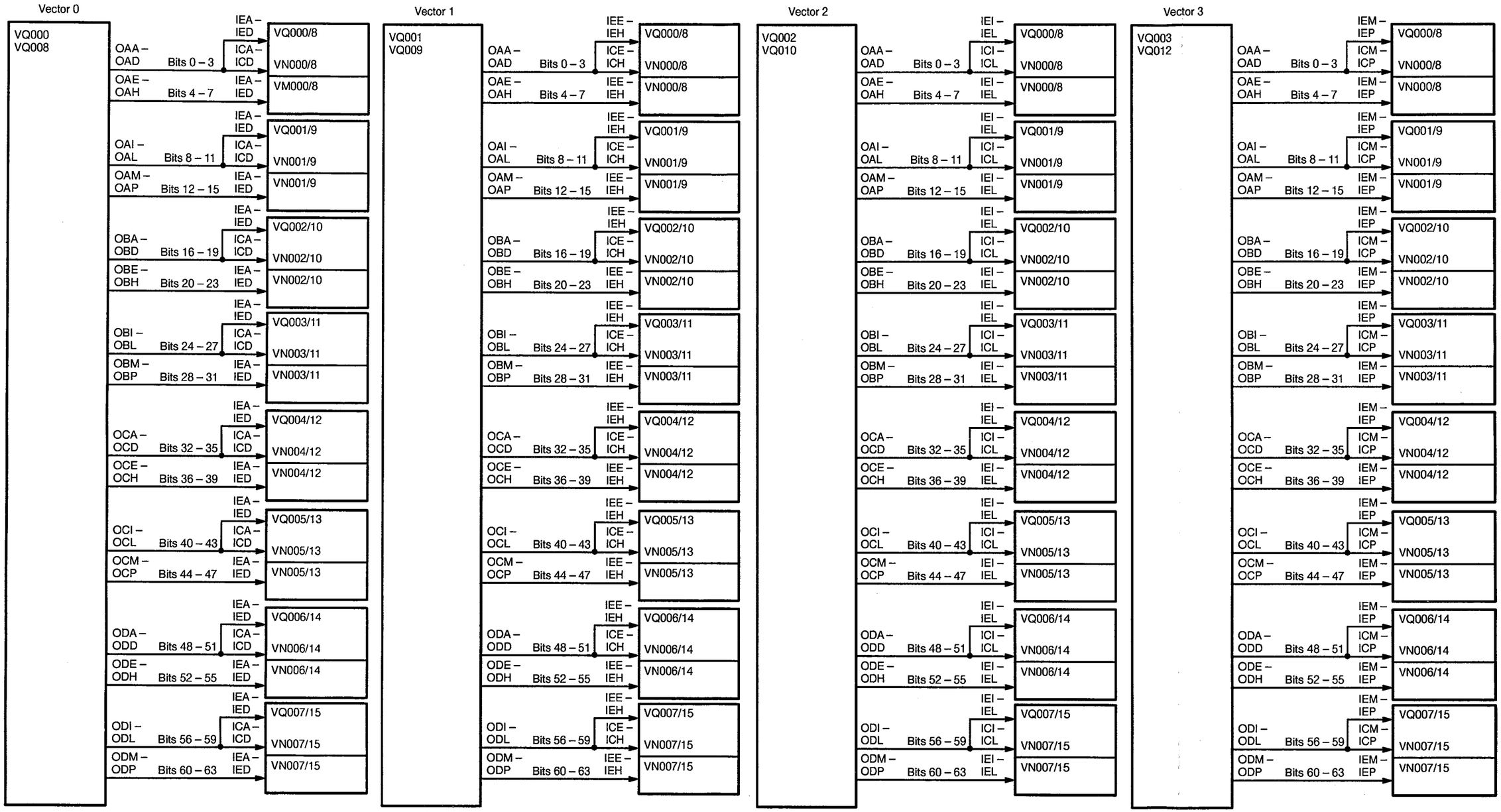


Figure 38. Vectors 0 through 3, Pipe 0/1, Read Data Path

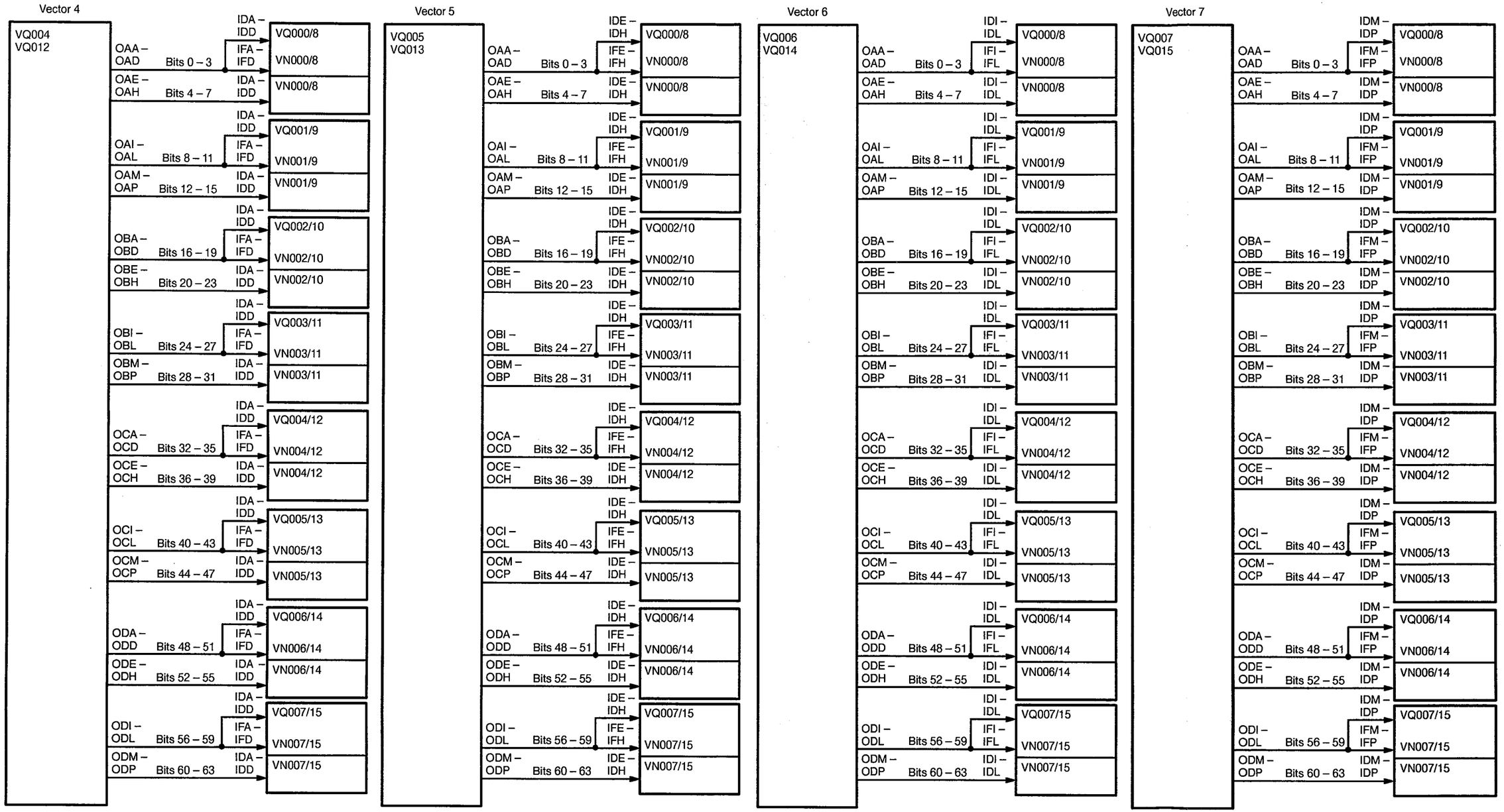


Figure 39. Vectors 4 through 7, Pipe 0/1, Read Data Path

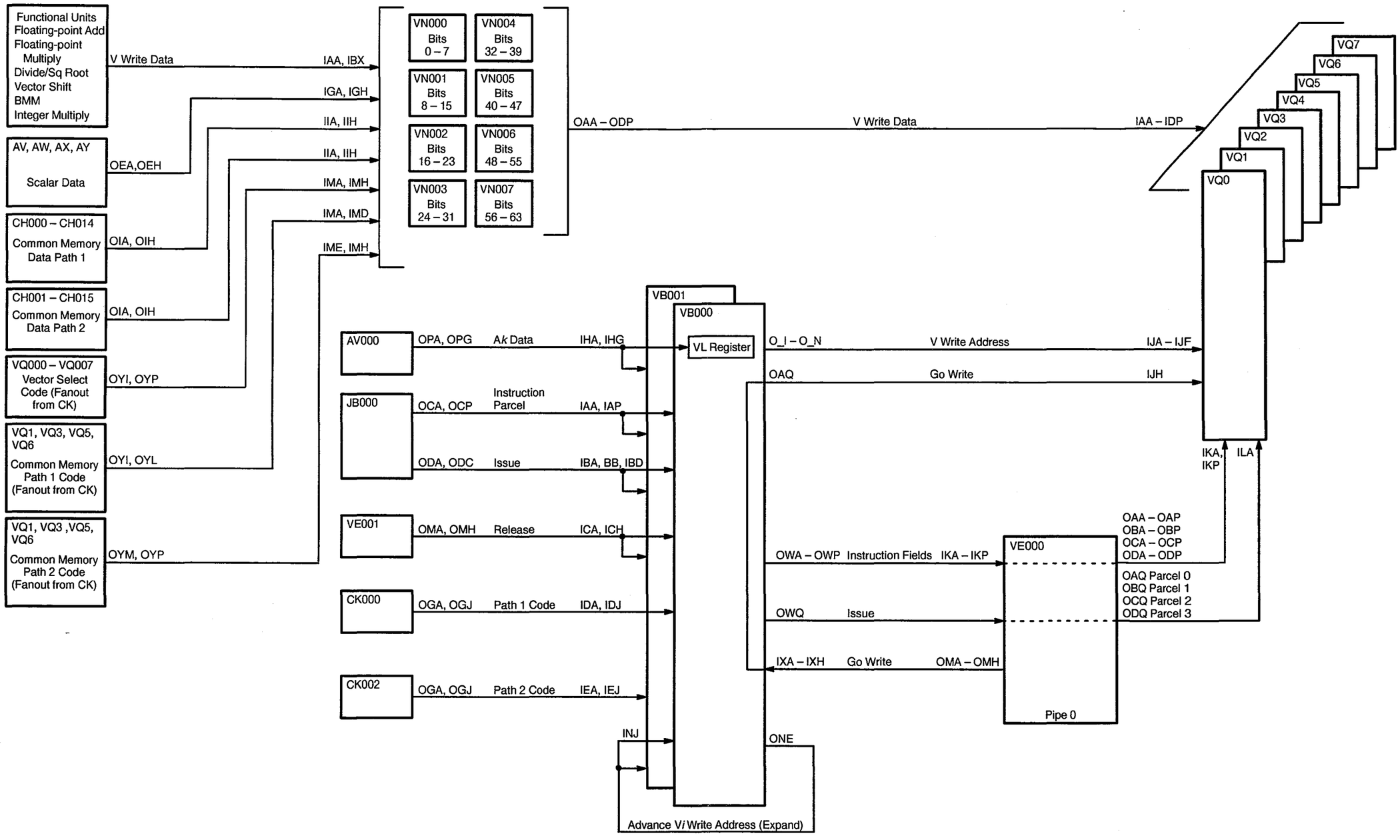
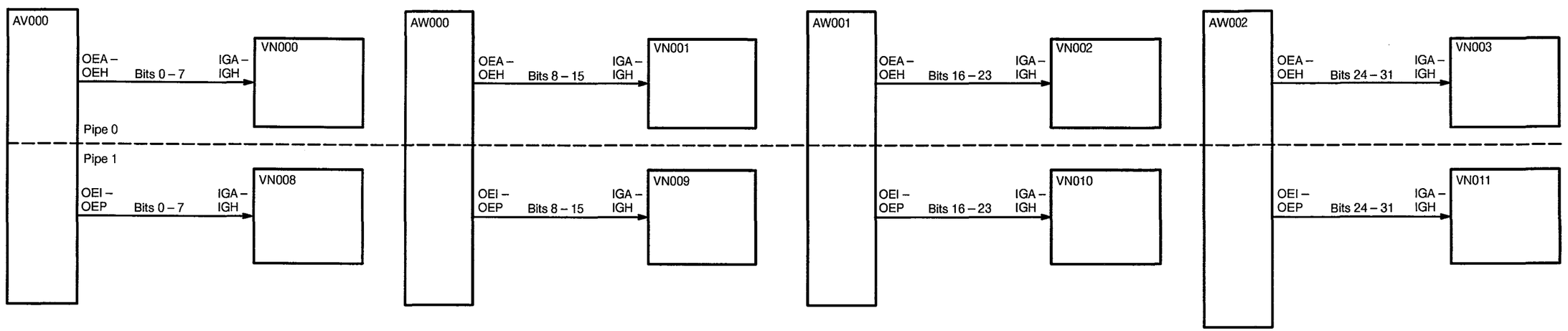


Figure 40. Vector Register Write Block Diagram, Pipe 0



S Register to Vector

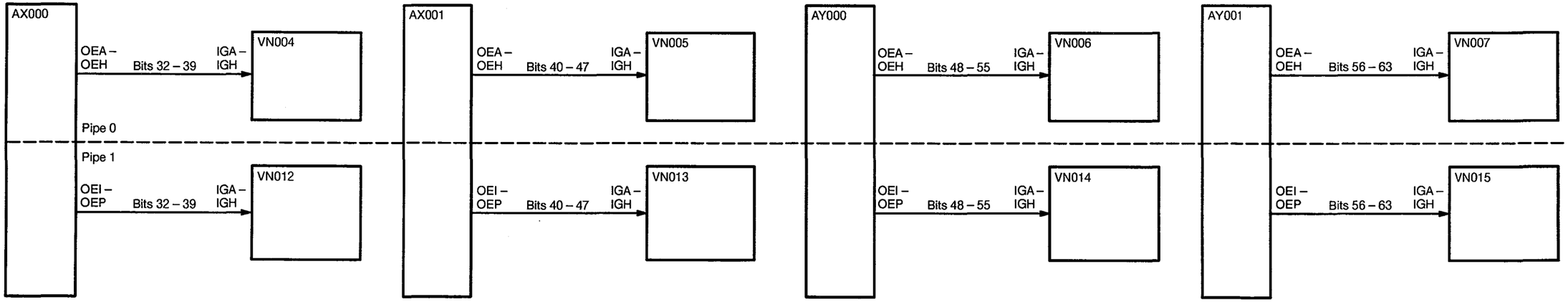
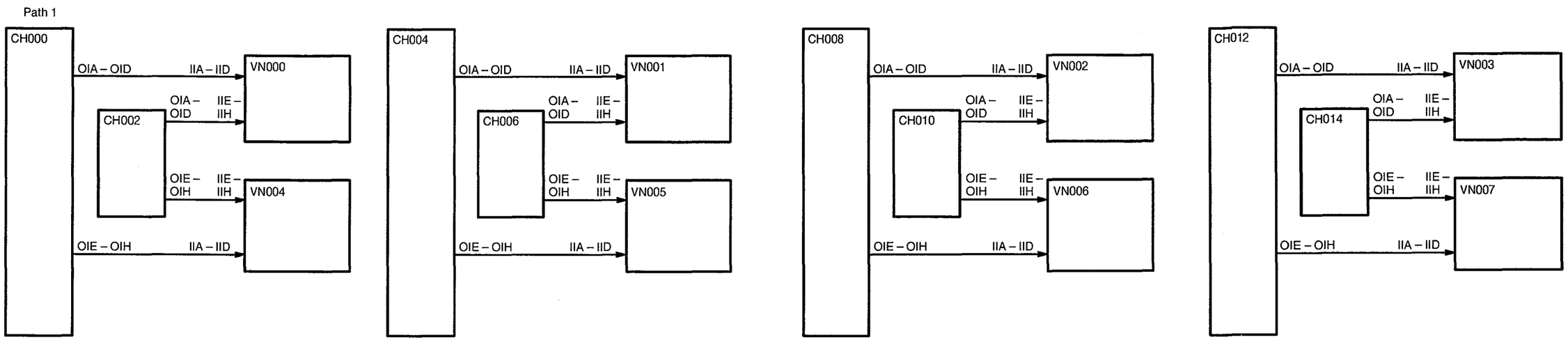
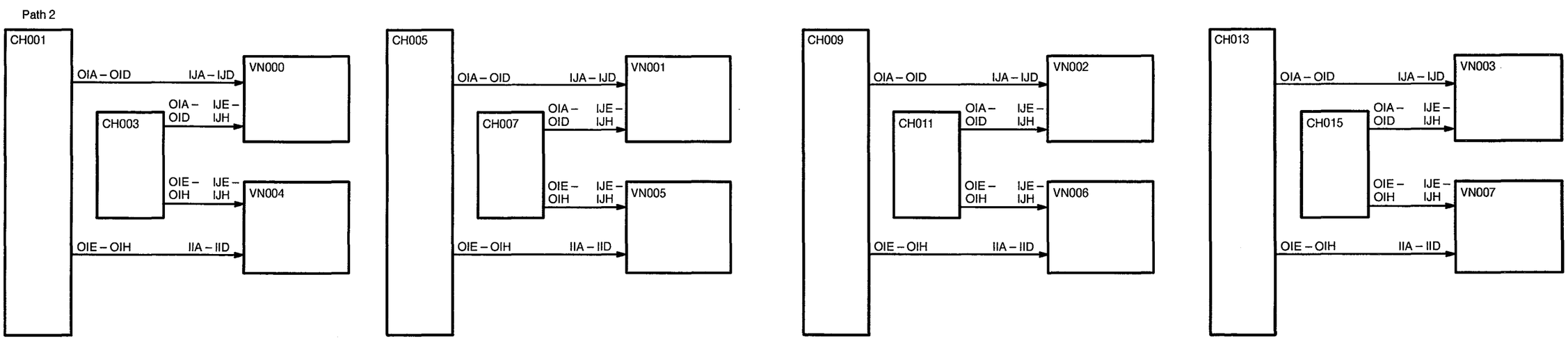


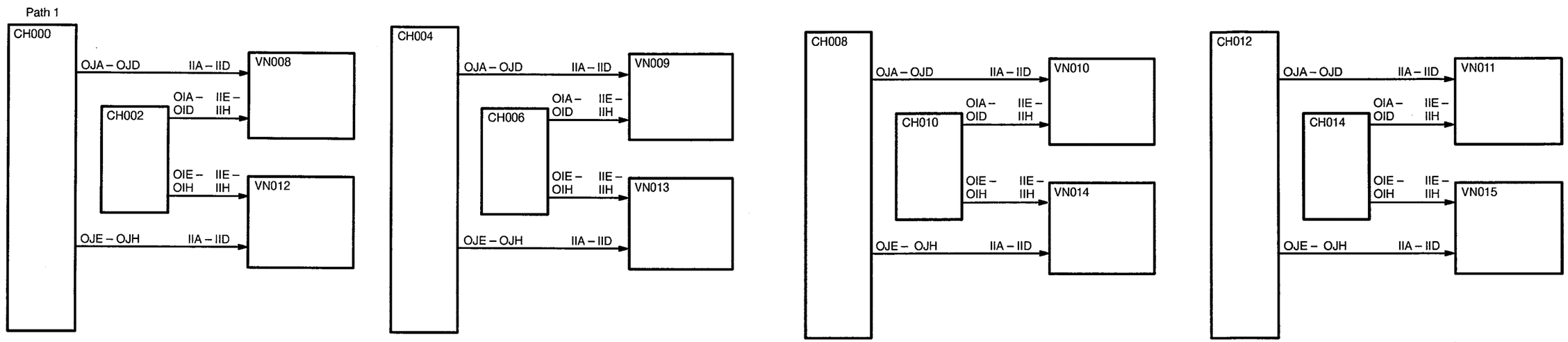
Figure 41. S Register to Vectors

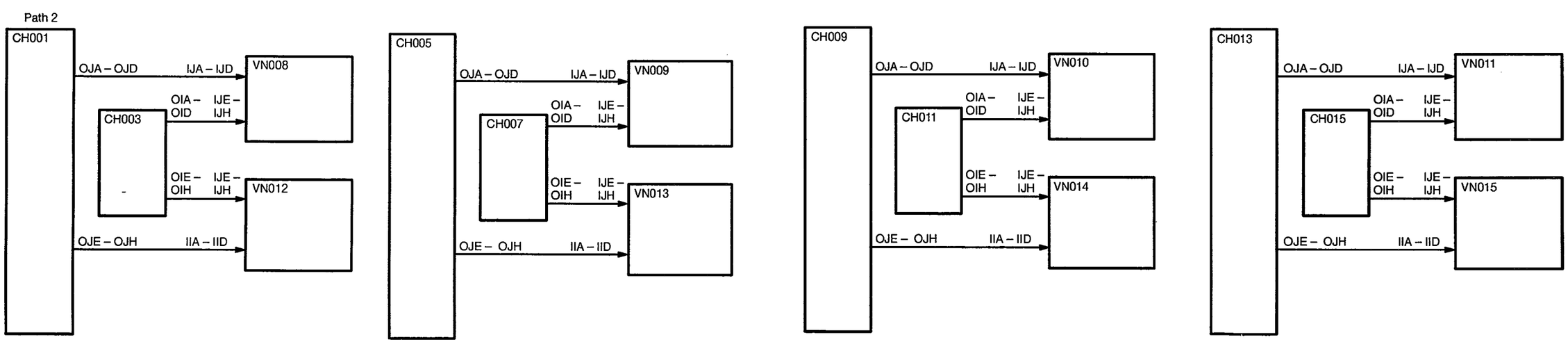




Common Memory Data to Vector Paths 1 and 2 Even Elements

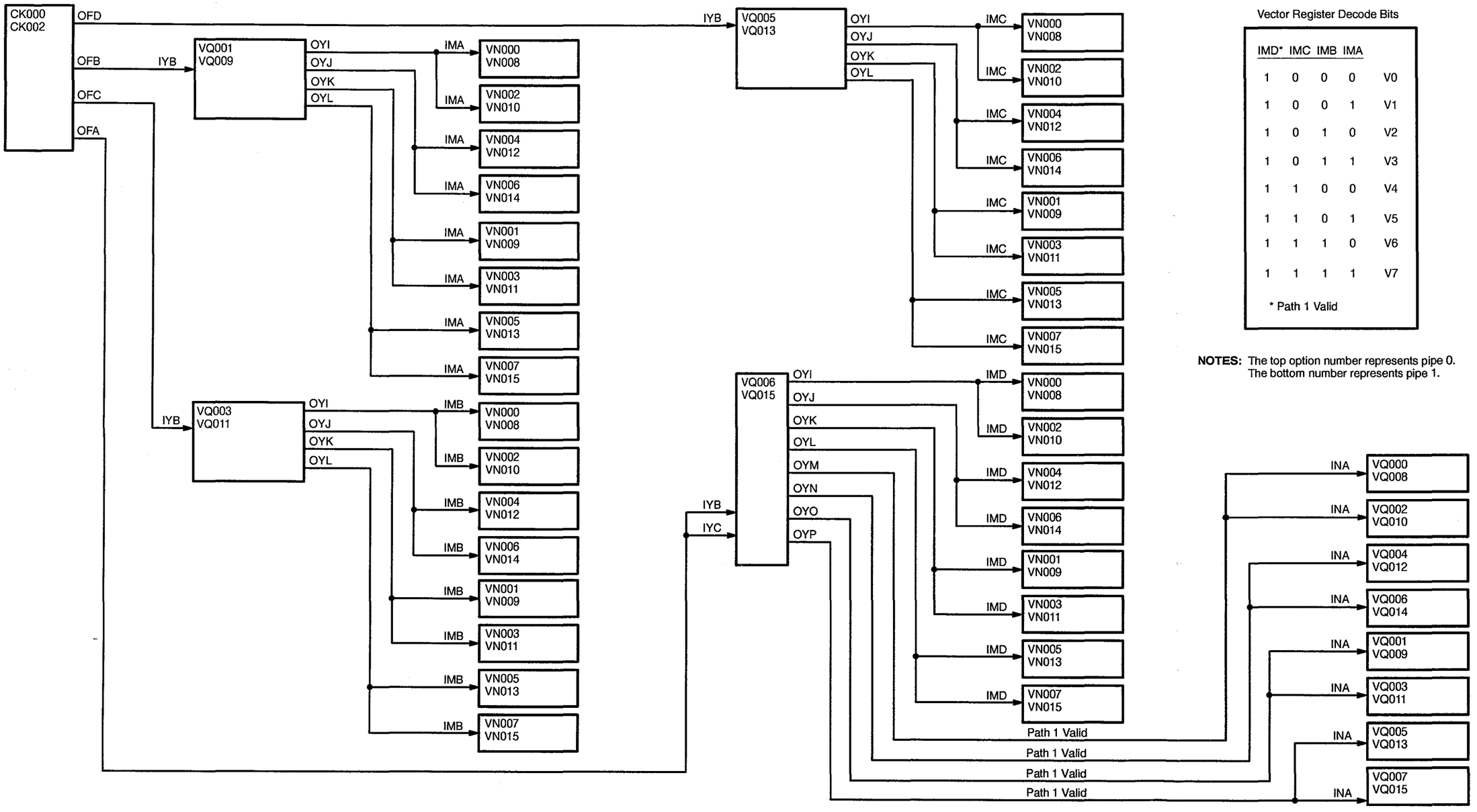
Figure 42. Memory Data to Vectors, Even Elements





Common Memory Data to Vector Paths 1 and 2 Odd Elements

Figure 43. Memory Data to Vectors, Odd Elements



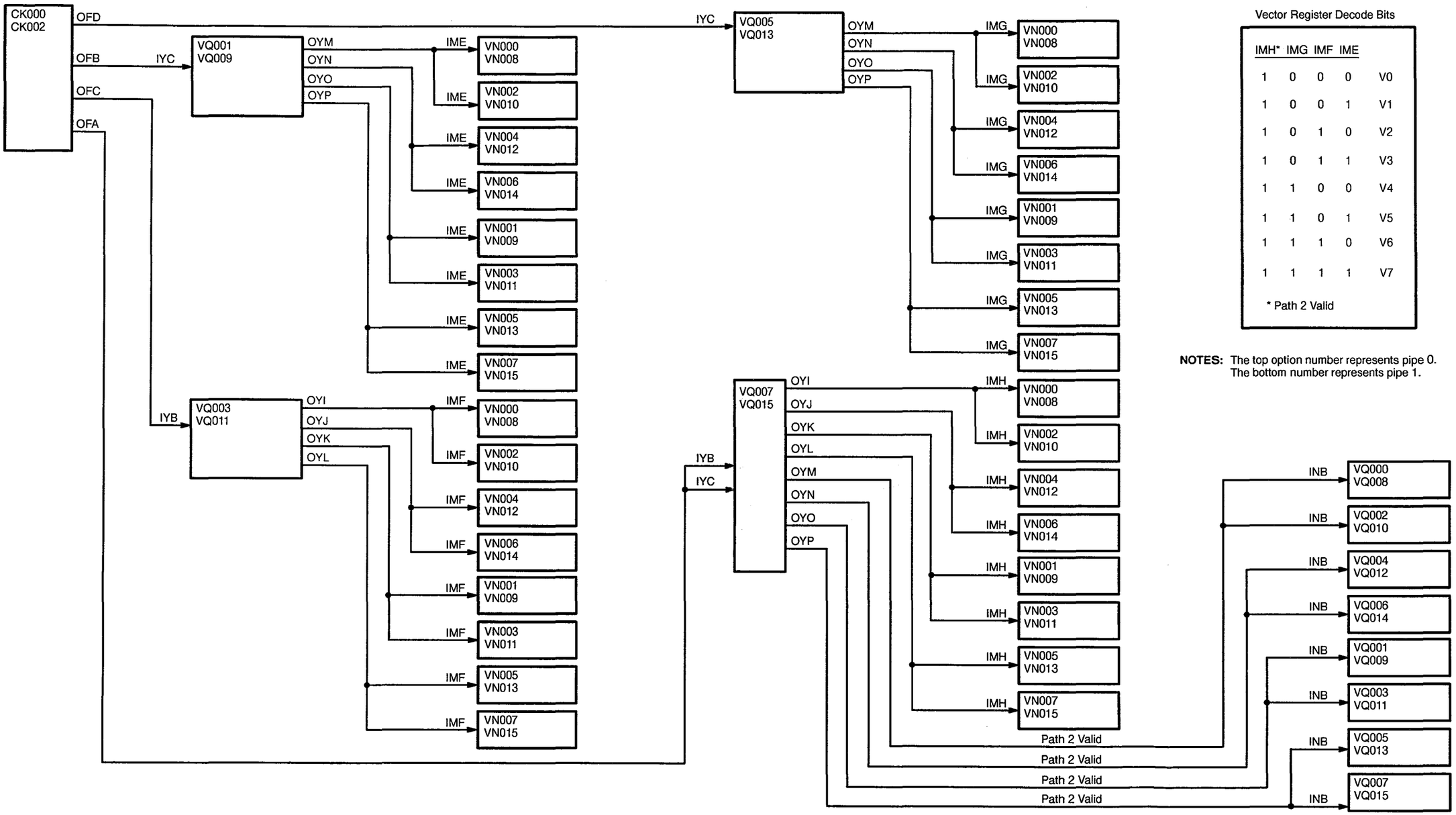
Vector Register Decode Bits

IMD*	IMC	IMB	IMA	
1	0	0	0	V0
1	0	0	1	V1
1	0	1	0	V2
1	0	1	1	V3
1	1	0	0	V4
1	1	0	1	V5
1	1	1	0	V6
1	1	1	1	V7

* Path 1 Valid

NOTES: The top option number represents pipe 0.
The bottom number represents pipe 1.

Figure 44. Vector Register Decode Bit Fanout, Pipe 0 and 1, Path 1 Only



Vector Register Decode Bits

IMH*	IMG	IMF	IME	
1	0	0	0	V0
1	0	0	1	V1
1	0	1	0	V2
1	0	1	1	V3
1	1	0	0	V4
1	1	0	1	V5
1	1	1	0	V6
1	1	1	1	V7

* Path 2 Valid

NOTES: The top option number represents pipe 0.
The bottom number represents pipe 1.

Figure 45. Vector Register Decode Bit Fanout, Pipe 0 and 1, Path 2 Only

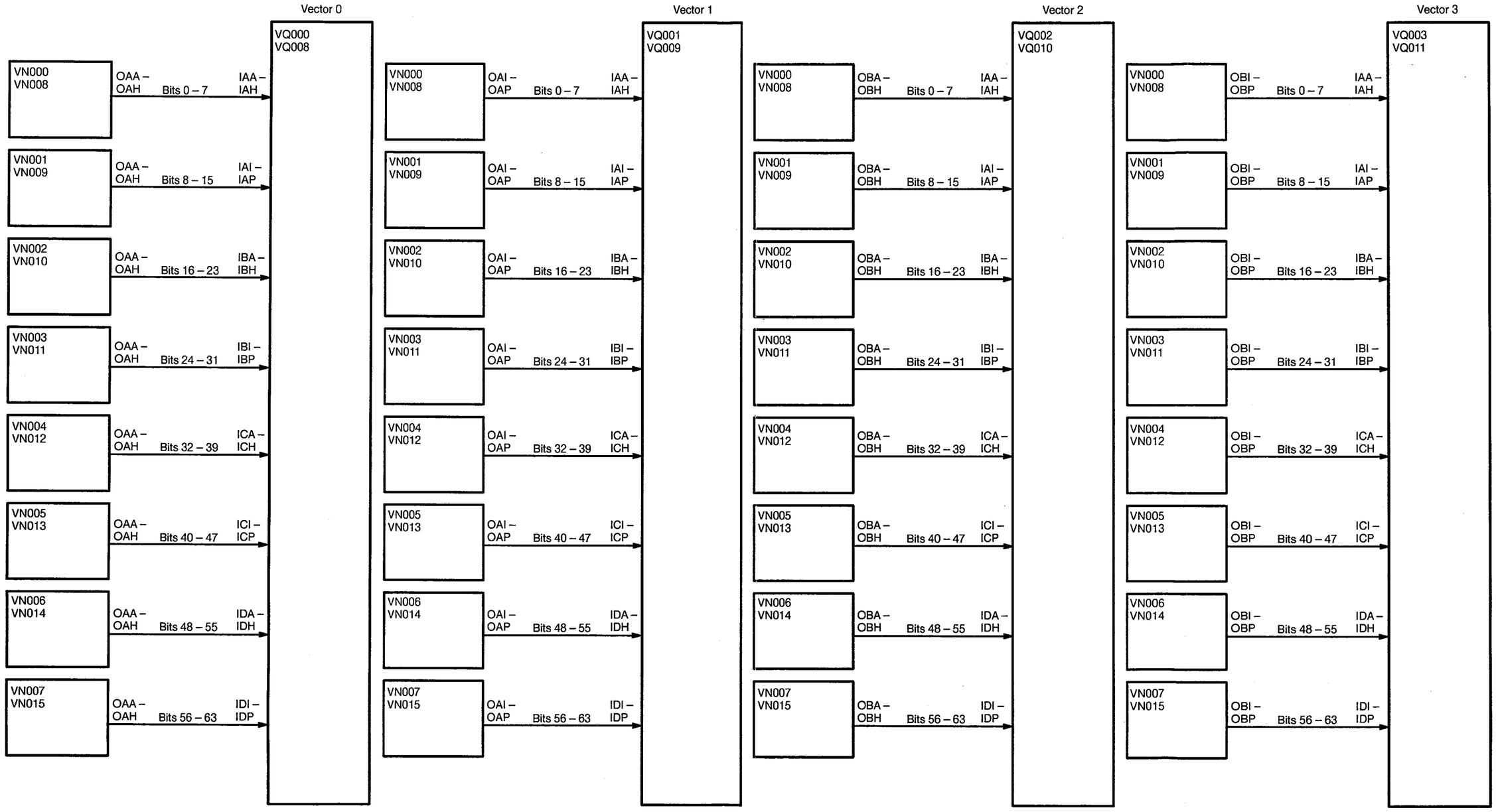


Figure 46. Vectors 0 through 3, Pipe 0/1, Write Data Path

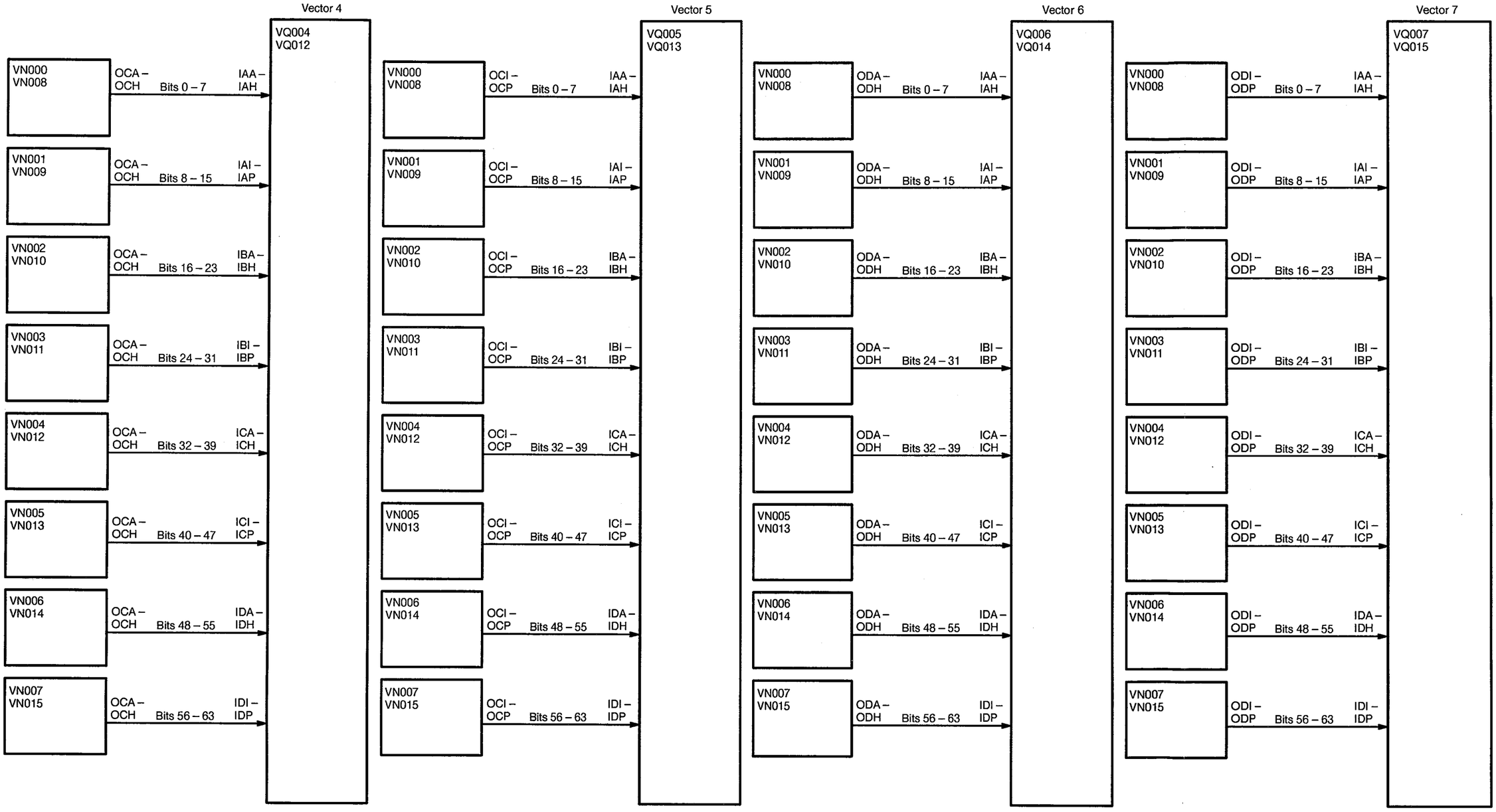


Figure 47. Vectors 4 through 7, Pipe 0/1, Write Data Path

VECTOR LOGICAL

There are two independent vector logical units in a CRAY T90 series system. (Refer to Figure 48 for a block diagram of the vector logical units.) These functional units reside on 16 VN options. VN000 through VN007 handle pipe 0 (the even elements), and VN008 through VN015 handle pipe 1 (the odd elements). Each VN option operates on a 4-bit slice of all eight vector registers.

The vector logical units receive input data from the VQ options and send the results 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 processed first in the second unit 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 [also called exclusive OR (XOR)] functions using either scalar or vector registers.

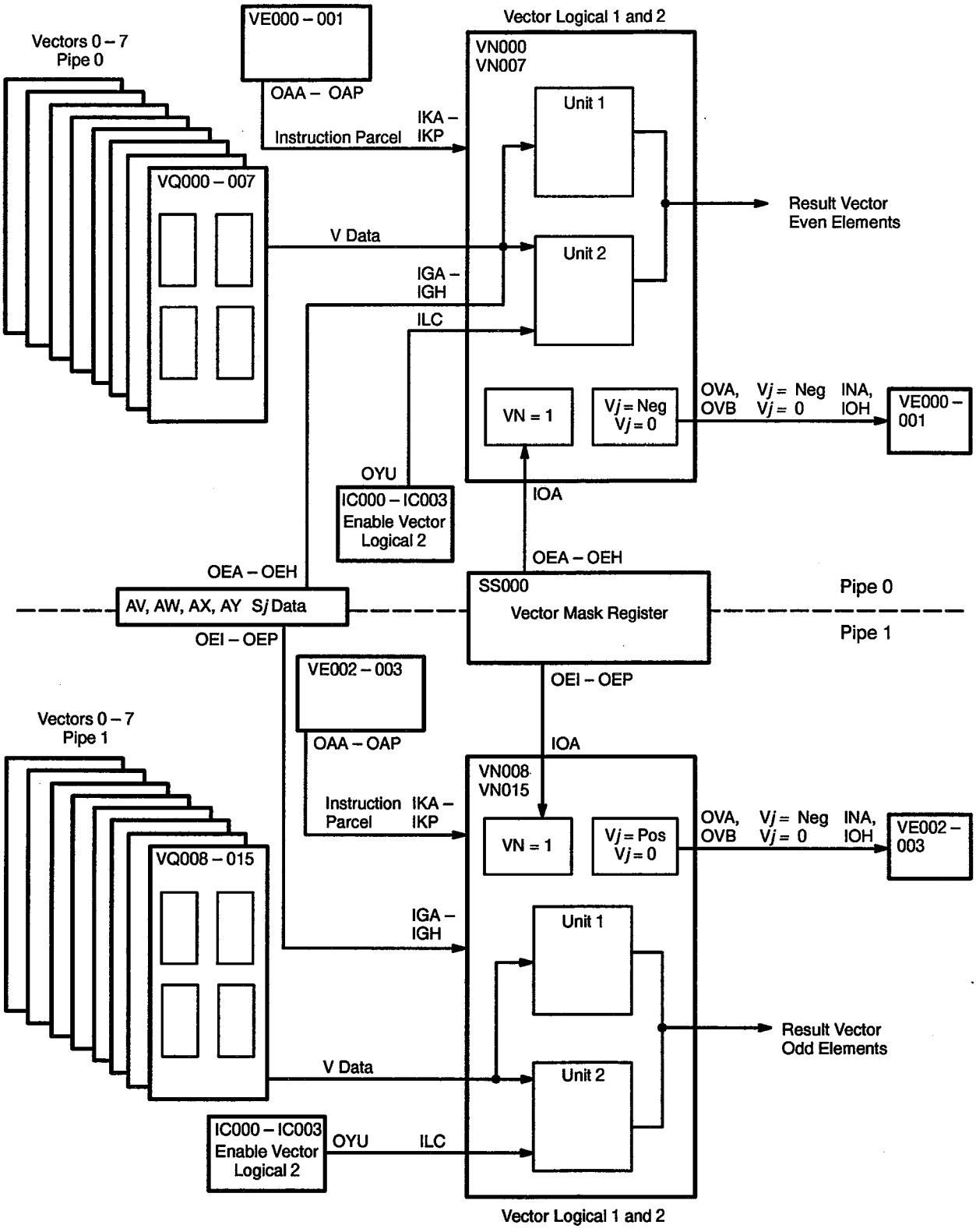


Figure 48. Vector Logical Block Diagram

Vector Logical Instructions

Table 15 lists the vector logical instructions.

Table 15. Vector Logical Instructions

Instruction	CAL	Description
140*ijk*	V*i* S*j*&V*k*	Transmit logical product of (S*j*) and (V*k* elements) to V*i* elements
141*ijk*	V*i* V*j*&V*k*	Transmit logical product of (V*j* elements) and (V*k* elements) to V*i* elements
142*ijk*	V*i* S*j*!V*k*	Transmit logical sum of (S*j*) and (V*k* elements) to V*i* elements
143*ijk*	V*i* V*j*!V*k*	Transmit logical sum of (V*j* elements) and (V*k* elements) to V*i* elements
144*ijk*	V*i* S*j*^V*k*	Transmit logical differences of (S*j*) and (V*k* elements) to V*i* elements
145*ijk*	V*i* V*j*^V*k*	Transmit logical differences of (V*j* elements) and (V*k* elements) to V*i* elements

Vector Merge

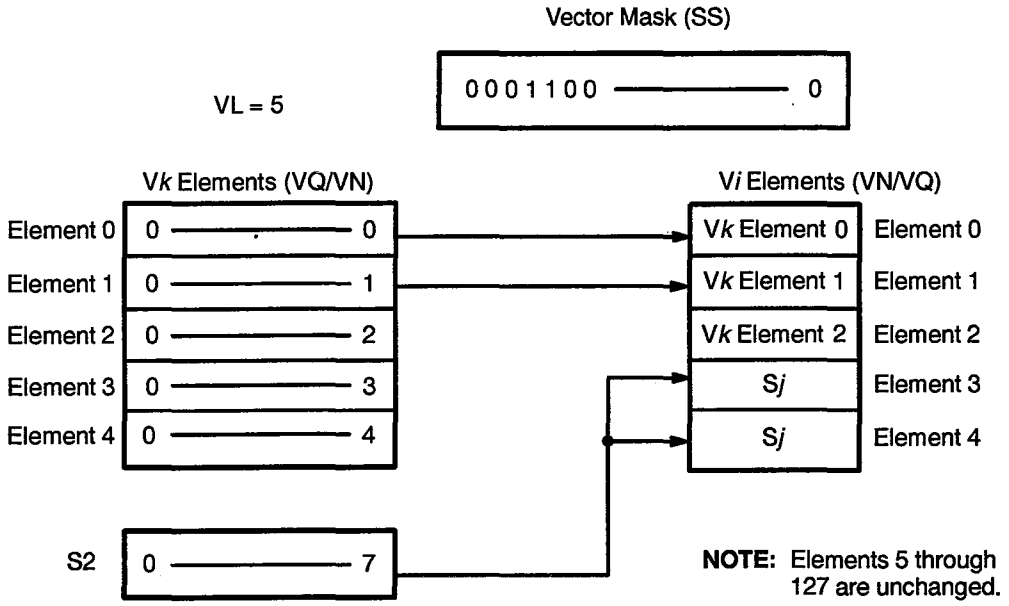
The 146 and 147 instructions merge the contents of the registers using the vector mask register for control. The 146 instruction merges the contents of S*j* with the contents of V*k*; the 147 instruction merges the contents of V*j* and V*k*. If the vector mask bit is a 1, the V*j* or S*j* data is used; 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
146*ijk*	V*i* S*j*!V*k*&VN	Merge (S*j*) and (V*k* elements) to V*i* elements using (VN) as mask
146*i*0*k*	V*i*#VN&V*k*	Merge 0 and (V*k* elements) to V*i* elements using (VN) as mask
147*ijk*	V*i* V*j*!V*k*&VN	Merge (V*j* elements) and (V*k* elements) to V*i* elements using (VN) as mask

147*ijk* Merge S*j* and V*k* elements to V*i* elements using VN as mask



146*ijk* Merge V*j* elements and V*k* elements to V*i* elements using VN as mask

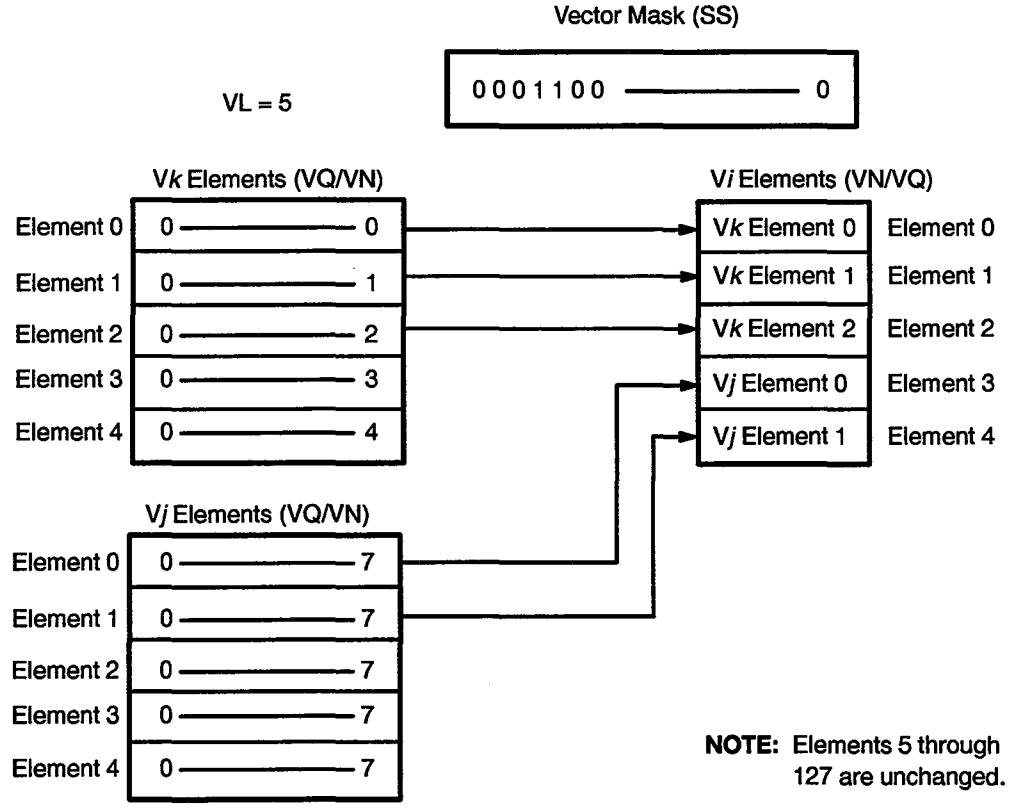


Figure 49. Vector Merge Operation

Vector Mask

VM0 and VM1 are vector mask registers. 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 the test condition of an element in a vector register. For example, the mask register can indicate which elements of a particular vector register contain positive values.

The vector mask register receives data either 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 1750*j*0 instructions.

Table 17. Vector Mask Operations

Instruction	CAL	Description
0030*j*0	VM0 S*j*	Transmit (S*j*) to VM0
0030*j*1	VM1 S*j*	Transmit (S*j*) to VM1
*0030*j*2	VM0 A*j*	Transmit (A*j*) to VM0
*0030*j*3	VM1 A*j*	Transmit (A*j*) to VM1
070*ij*1	V*i* CI,S*j*&VM	Transmit compressed index of (S*j*) controlled by (VM) to V*i*
073*i*00	S*i* VM0	Transmit (VM0) to S*i*
073*i*10	S*i* VM1	Transmit (VM1) to S*i*
*073*i*20	A*i* VM0	Transmit (VM0) to A*i*
*073*i*30	A*i* VM1	Transmit (VM1) to A*i*

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

Table 18. Vector Mask Test Operations

Instruction	CAL	Description
1750*j*0	VM V*j*,Z	Set VM bit if (V*j* element) = 0
1750*j*1	VM V*j*,N	Set VM bit if (V*j* element) ≠ 0
1750*j*2	VM V*j*,P	Set VM bit if (V*j* element) ≧ 0
1750*j*3	VM V*j*,M	Set VM bit if (V*j* element) < 0
175*ij*4	V*i*,VM V*j*,Z	Set VM bit if (V*j* element) = 0 and store compressed indices of V*j* elements = 0 in V*i*
175*ij*5	V*i*,VM V*j*,N	Set VM bit if (V*j* element) ≠ 0 and store compressed indices of V*j* elements ≠ 0 in V*i*
175*ij*6	V*i*,VM V*j*,P	Set VM bit if (V*j* element) ≧ 0 and store compressed indices of V*j* elements ≧ 0 in V*i*
175*ij*7	V*i*,VM V*j*,M	Set VM bit if (V*j* element) < 0 and store compressed indices of V*j* elements < 0 in V*i*

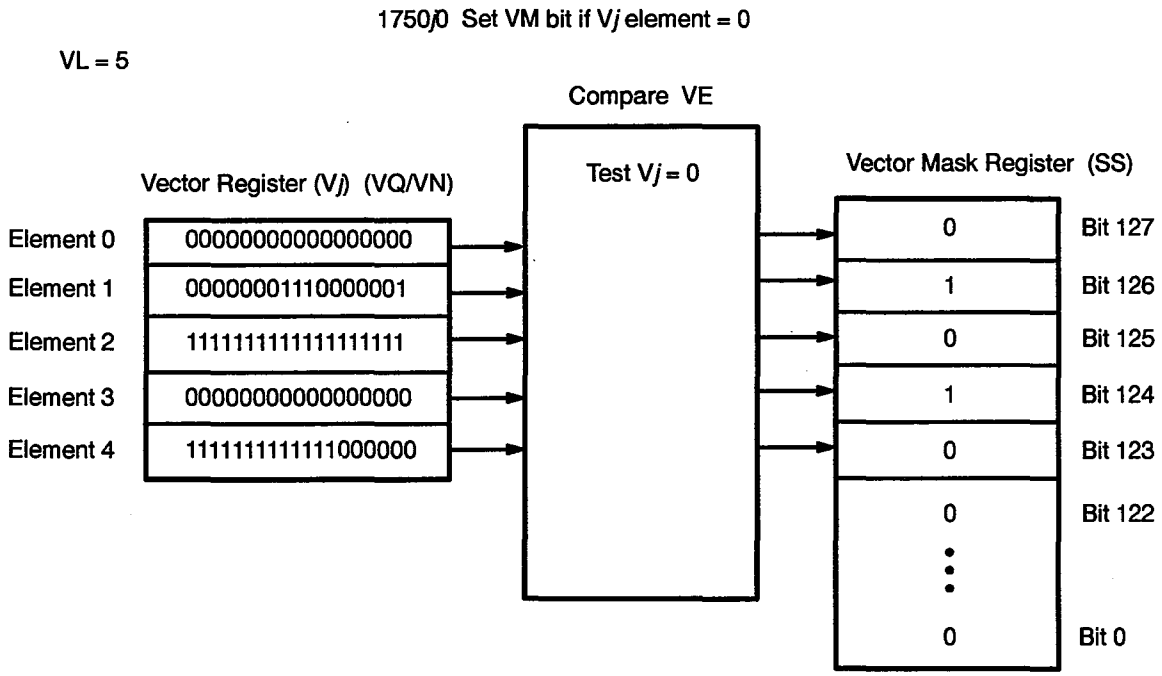


Figure 50. 1750*j*0 Instructions

Figure 51 illustrates the function of the 175*ij*4 instructions that use the vector mask to create a compressed vector.

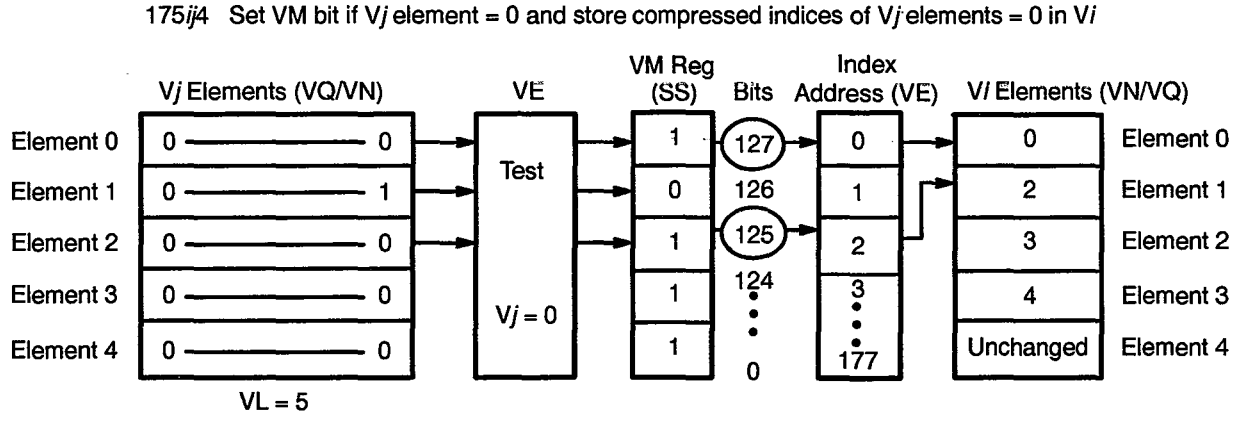


Figure 51. Function of the 175*ij*4 Instructions

Compressed Iota

The iota function is performed on the RE options. RE000 is used for pipe 0 and RE001 is used for pipe 1. Table 19 lists the instruction used in iota operations, and Figure 52 is a block diagram of iota pipe 0 and 1.

Table 19. Iota Instruction

Instruction	CAL	Description
070*ij*1	V*i* CI,S*j*&VM	Transmit compressed index of (S*j*) controlled by (VM) to V*i*

The 070*ij*1 instruction forms multiples of the contents of register S*j* starting with 0 (0, S*j*, 2 x S*j*, 3 x S*j*, and so on). It stores multiples that correspond to each 1 bit set in the vector mask register in successive elements of register V*i* (beginning at element 0). The instruction stops when all unused bits of the vector mask are 0 or are used.

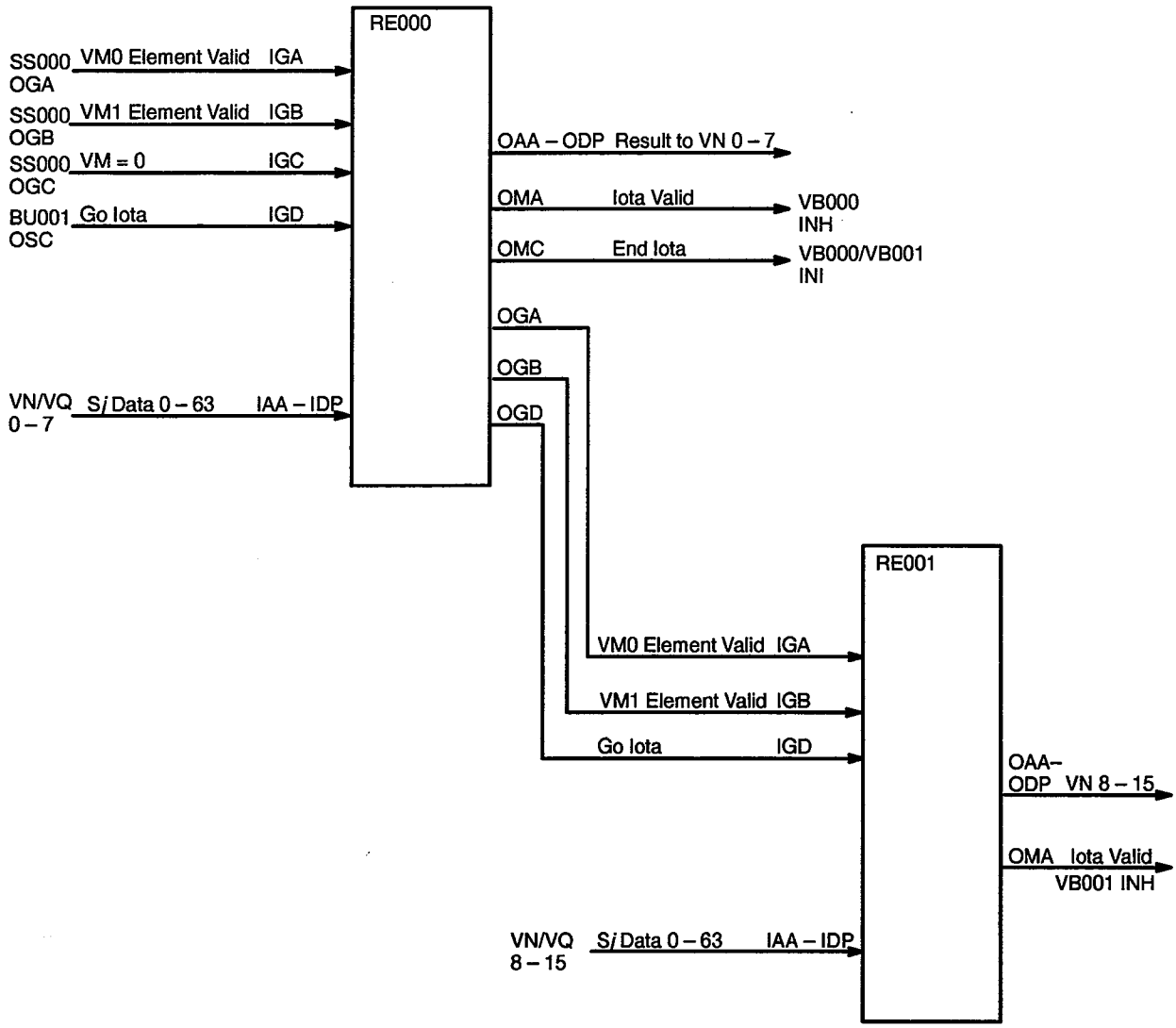


Figure 52. Iota Pipe 0 and 1

Figure 53 on page 100 illustrates the function of the 070*ij*1 instructions that use the vector mask to create a compressed vector.

RE Option

The RE000 receives the Go Iota signal from the BU001 option, makes a copy of this signal, and sends it to the RE001 option. The S*j* data arrives at both options along with a Element Valid signal. After the operand has been used and a pair of elements is ready to be written to the result vector,

the Iota Valid signal is sent to the VB option. The two Iota Valid signals, one from RE000 and on from RE001, are usually identical except when there is an odd number of elements on Pipe 0. The operation ends when the VM=0 signal arrives from the SS option and causes the RE000 option to send the Signal End Iota signal to both VB options. The Signal End Iota signal is sent concurrently with the last Element Valid signal.

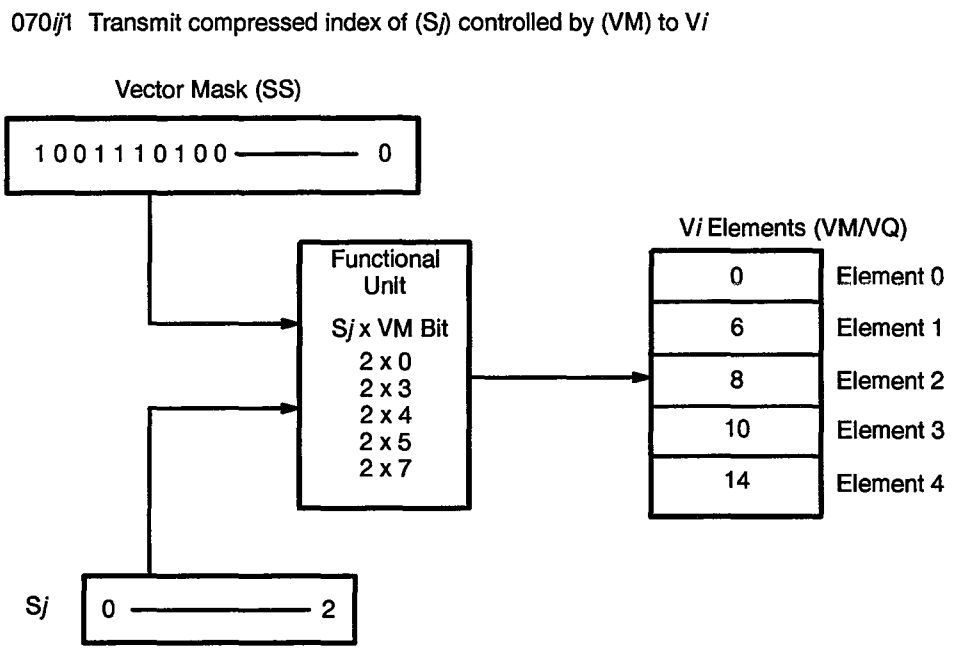


Figure 53. Function of the 070*ij*1 Instructions

VECTOR ADD

The vector add functional unit is located on the VN and VE options. (Refer to Figure 54 for a block diagram of vector add.) The VN options perform the actual addition of the input operands and then pass the group carries and group enables to the VE for summation. These bit toggles are then returned to the VN 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.

Table 20. Vector Add Instructions

Instruction	CAL	Description
154*ijk*	V*i* S*j*+V*k*	Transmit integer sum of (S*j*) and (V*k* elements) to V*i* elements
155*ijk*	V*i* V*j*+V*k*	Transmit integer sum of (V*j* elements) and (V*k* elements) to V*i* elements
156*ijk*	V*i* S*j*–V*k*	Transmit integer difference of (S*j*) and (V*k* elements) to V*i* elements
156*i*0*k*	V*i*–V*k*	Transmit two's complement of (V*k* elements) to V*i* elements
157*ijk*	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 content of the S*j* register as an input operand. The VN option keeps a copy of the S*j* register, which enables a subsequent instruction to proceed and change the content of S*j* without affecting the 154 or 156 instruction in progress.

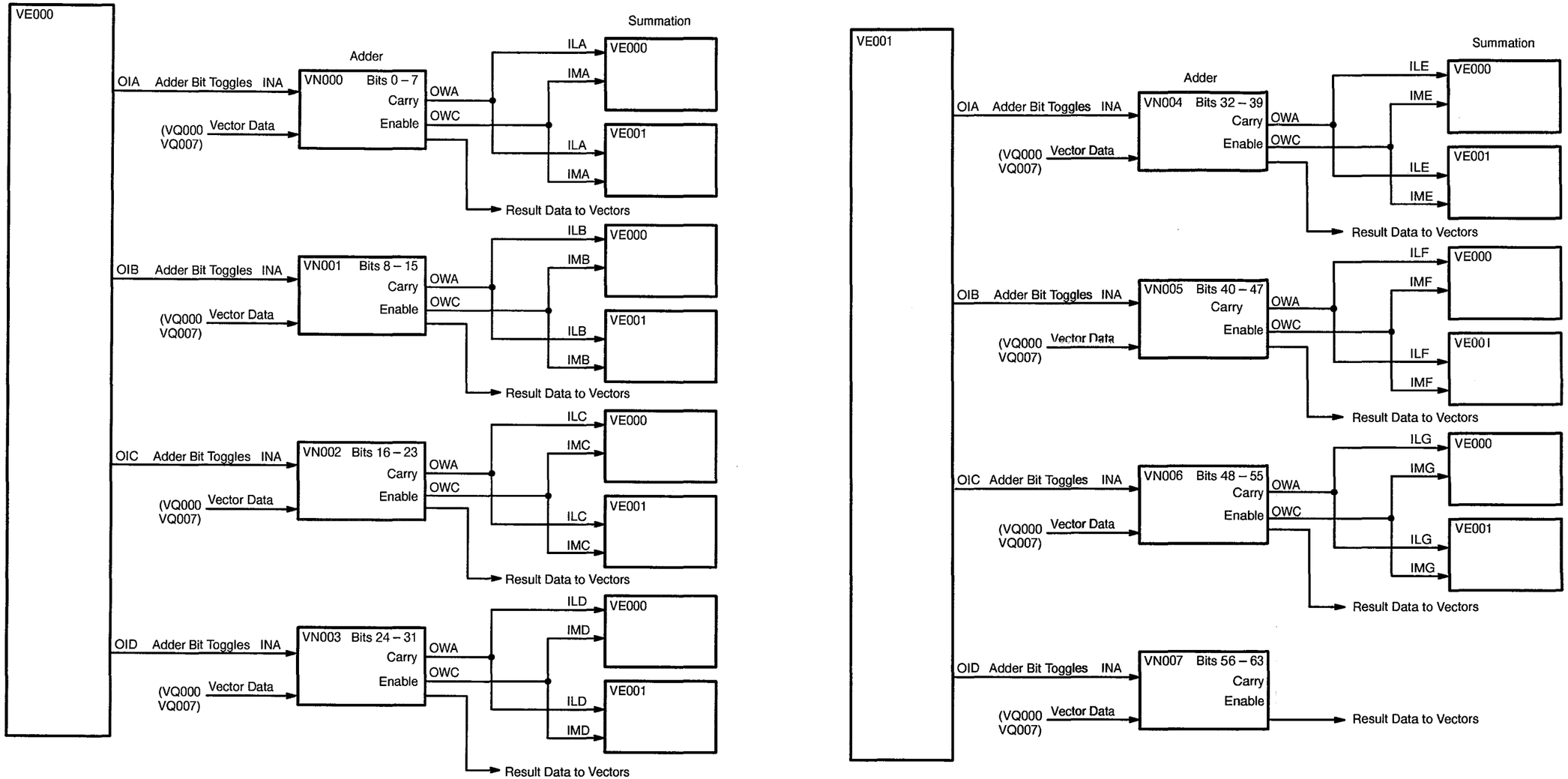


Figure 54. Vector Add Block Diagram

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 for each CPU.

The vector shift functional unit is responsible for vector transfer operations. For example, it transfers 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 V0 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
150*ijk*	V*i* V*j*<A*k*	Shift (V*j* elements) left (A*k*) places to V*i* elements
*150*ij*0	V*i* V*j*<V0	Shift (V*j* elements) left (V0 elements) places to V*i* elements
151*ijk*	V*i* V*j*>A*k*	Shift (V*j* elements) right (A*k*) places to V*i* elements
*151*ij*0	V*i* V*j*>V0	Shift (V*j* elements) right (V0 elements) places to V*i* elements
152*ijk*	V*i* V*j*,V*j*<A*k*	Double shift (V*j* elements) left (A*k*) places to V*i* elements
*152*ijk*	V*i* V*j*,A*k*	Transfer (V*j* elements) starting at element (A*k*) to V*i* elements
153*ijk*	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
*153*ij*0	V*i* V*j*,{VN]	Compress V*j* by (VN) to V*i*
*153*ij*1	V*i*,[VN] V*j*	Expand V*j* by (VN) 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 AV000 option, and all eight A series options check the value of the 64-bit A register. This test 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. AV000 sends bits 0 through 6 as the shift count.

To better understand this process, examine the composition of the shift count. For both single and double shifts, the shift count is similar except that the double shift has 1 extra bit (bit 6). Refer to Figure 55 for an examination 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. 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 and clears the result register.

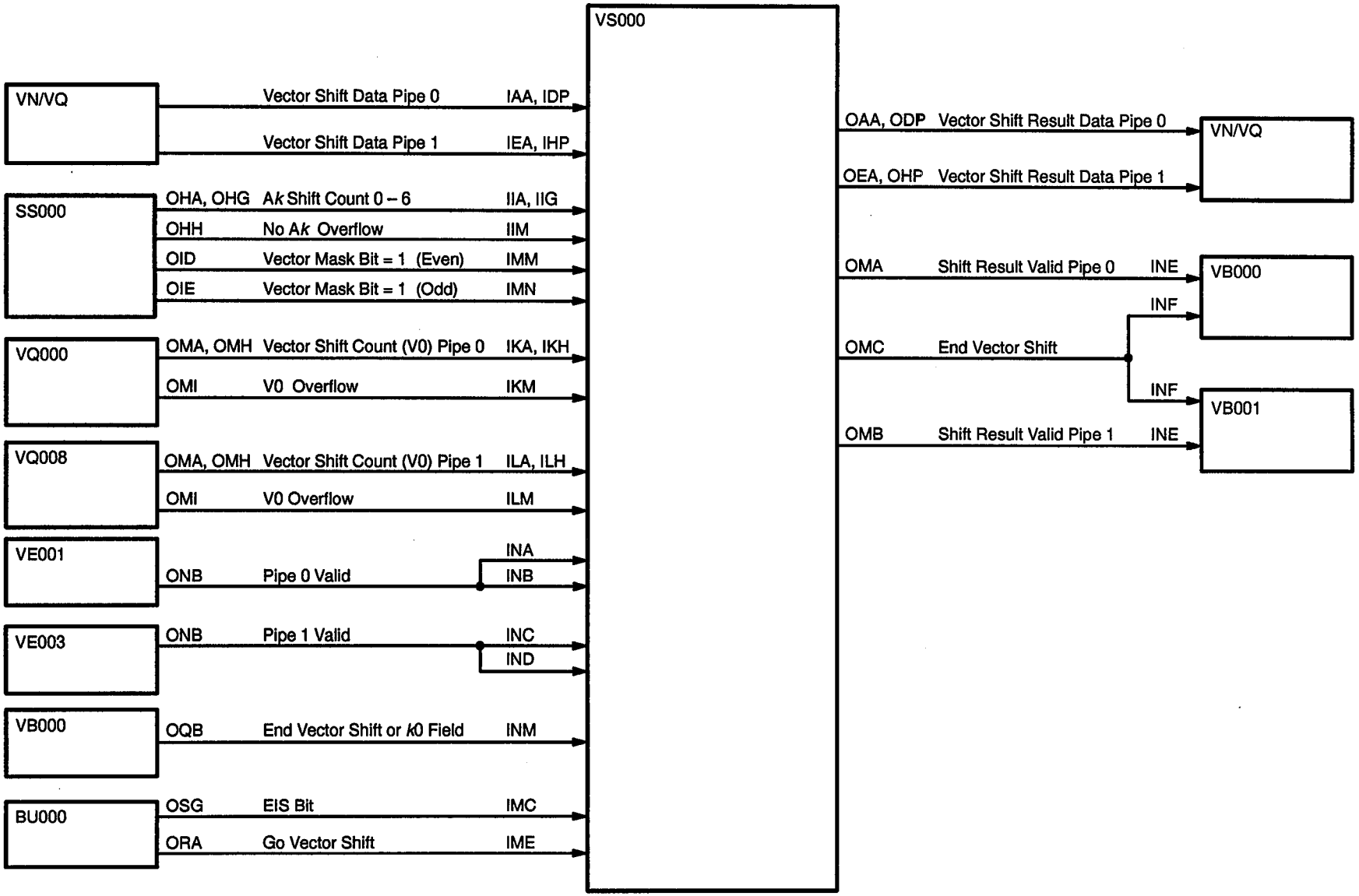


Figure 56. Vector Shift Block Diagram

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. In fact, the hardware performs only left shifts. Right shifts are accomplished according to the way data is entered into the shifter and the use of two's complement shift counts 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 VQ0 for pipe 0 and from VQ8 for pipe 1.

Vector Right Shift 005400 151*ij*0

Refer to Figure 57 for an illustration of a vector right shift using V0 for the shift count. Note that the shift count for element 0 is 0, which 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.

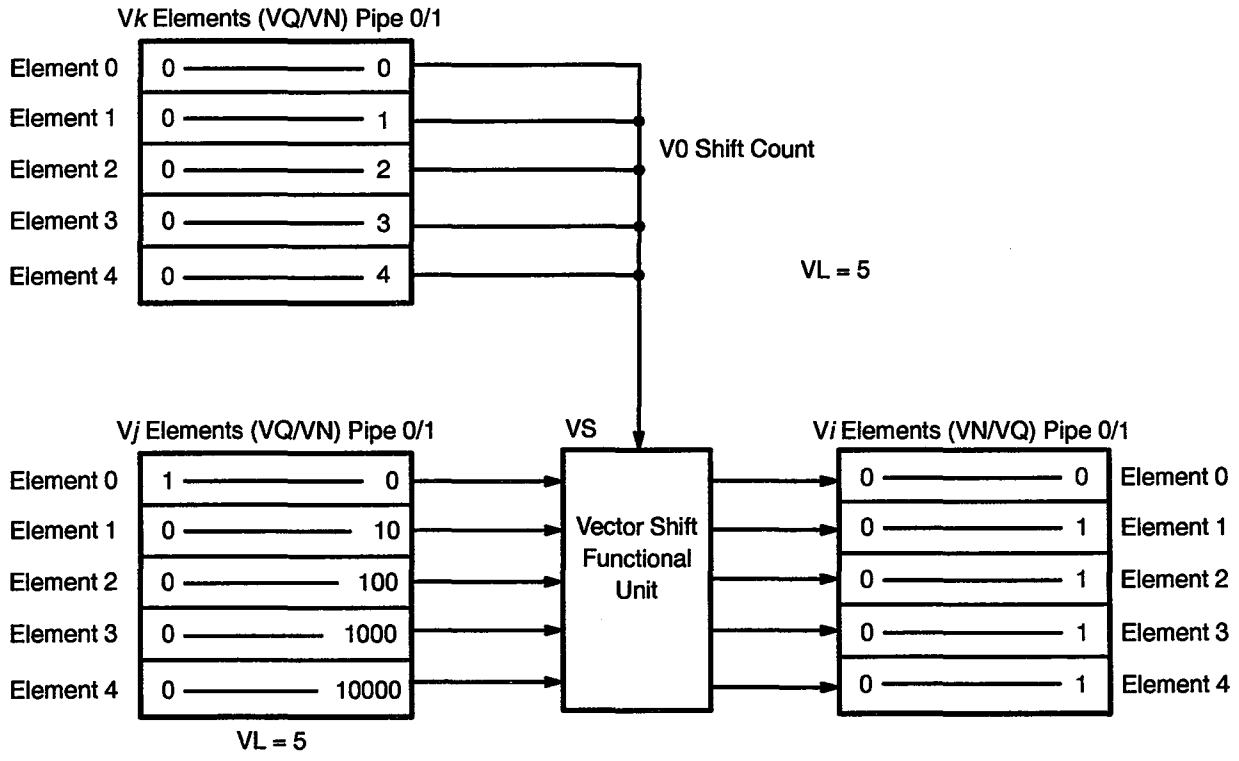


Figure 57. Vector Right Shift

Vector Right Double Shift 153*ijk*

Refer to Figure 58 for an illustration 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 containing 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, which results in an end-off shift for that element.

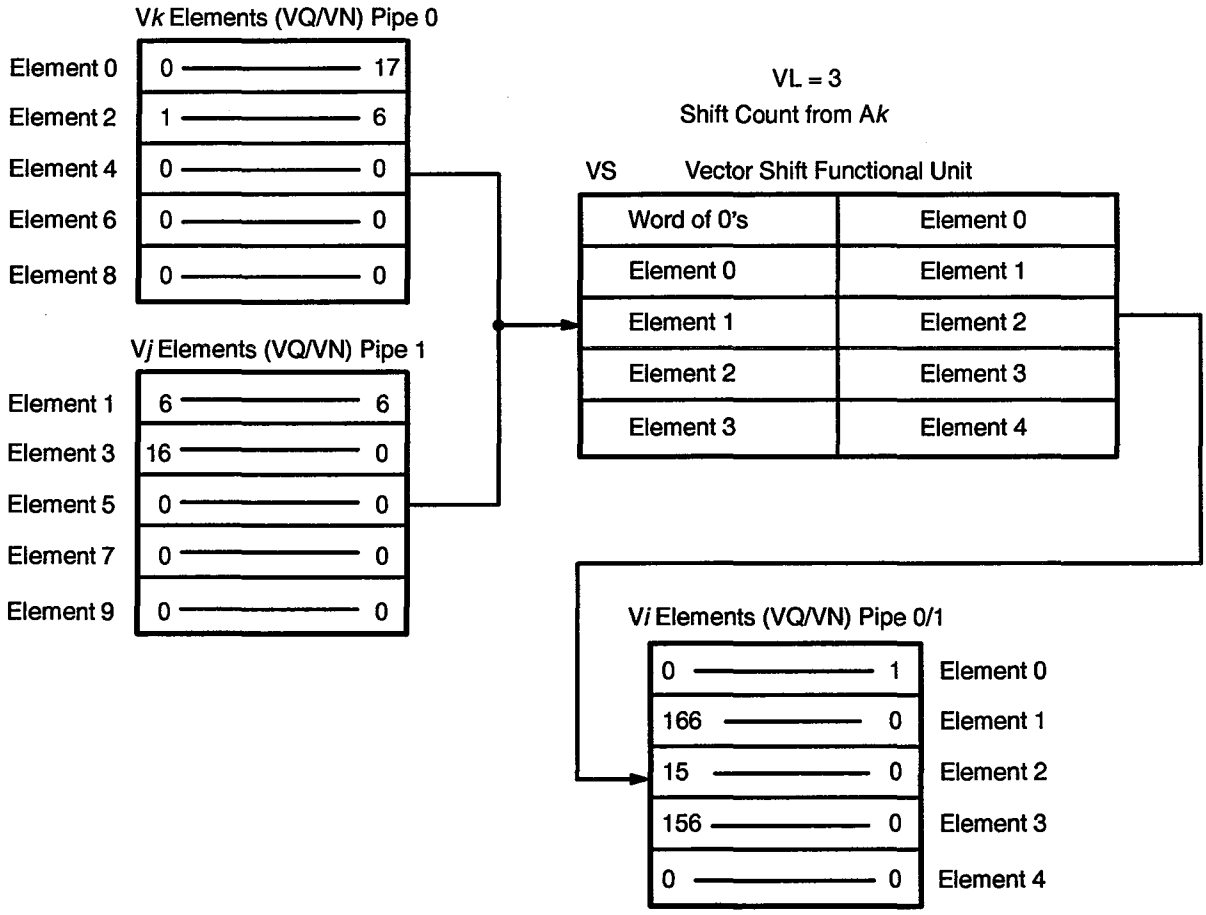


Figure 58. Vector Right Double Shift

Vector Transfer 005400 152*ijk*

This instruction moves the contents of V*j* to V*i* starting with element A*k* as illustrated in Figure 59. Note that this is an EIS instruction.

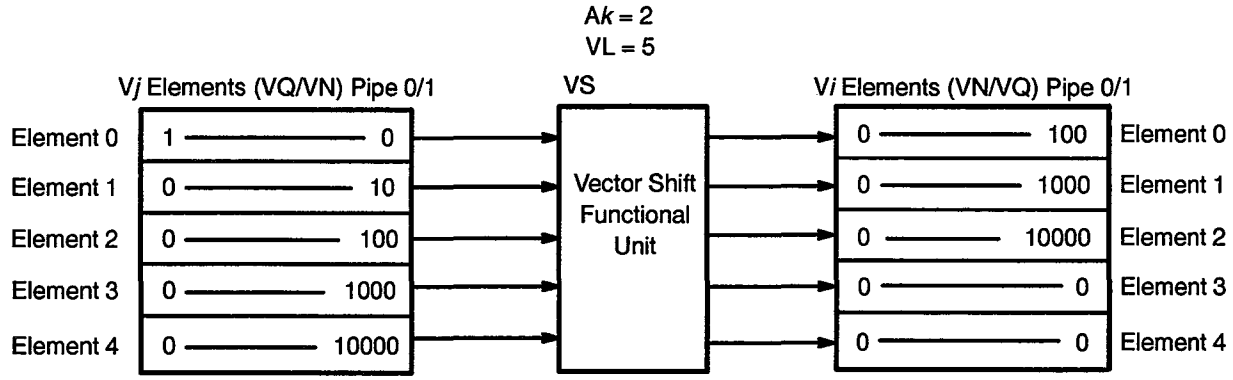


Figure 59. Vector Transfer

Vector Compress 005400 153*ij*0

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. 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.

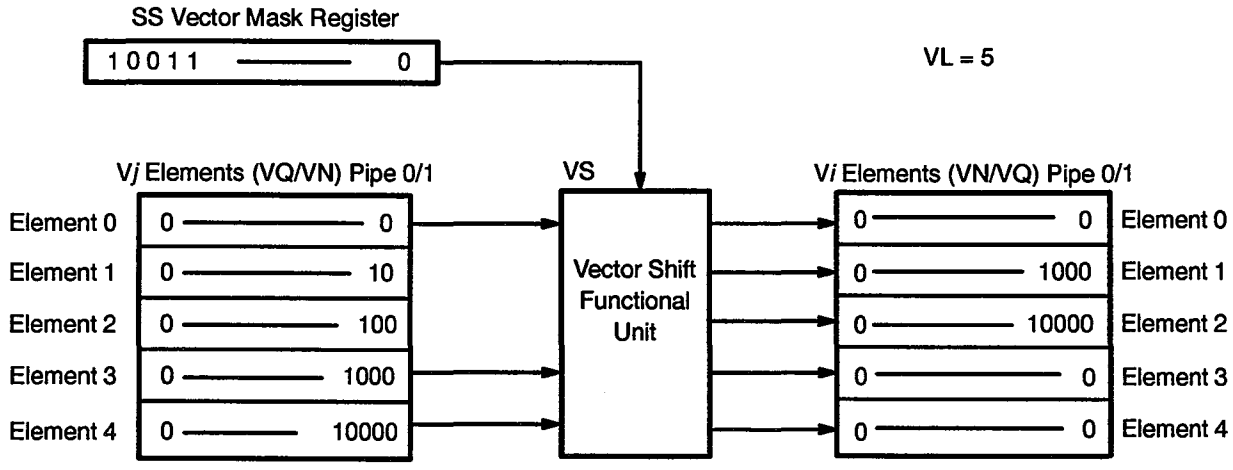


Figure 60. Vector Compress

Vector Expand 005400 153*ij*1

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.

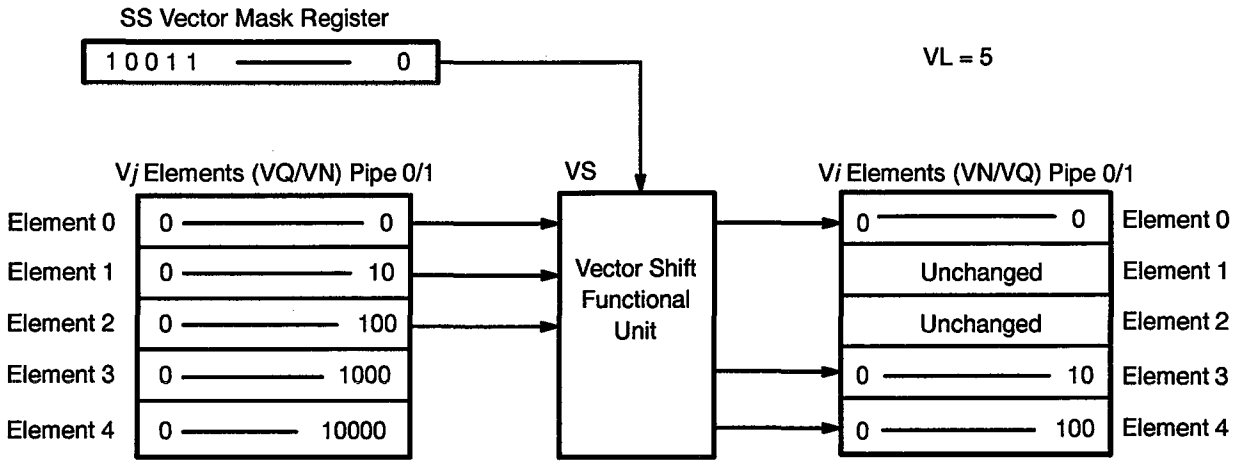


Figure 61. Vector Expand

VECTOR POP/ POP PARITY AND LEADING ZERO

The vector population/parity functional unit performs the population count (174*ij*1) and parity for vector operations (174*ij*2) instructions. This functional unit shares logic with the Divide and Square Root functional unit. The *k* field of the instruction determines the type of operation to be performed. Refer to Figure 62 for a block diagram of the vector population/parity functional unit.

The vector population/parity functional unit shares logic with the divide/square root functional unit. Therefore all vector operations reserve the associated functional unit. The divide/square root functional unit is reserved when the vector population/parity functional unit is reserved and vice versa.

Both scalar and vector register operations share the divide/square root functional unit. Therefore, when vector divide/square root, or vector population/parity instructions are executed, a scalar divide/square root instruction must wait until the vector operation is finished.

The 174*ij*1 instruction counts the number of 1 bits in each element of a vector register specified by V*j*. Each element is counted individually, and the result is stored in the corresponding 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 specified by the vector length.

The 174*ij*2 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 174*ij*2 instruction uses the same logic as the 174*ij*1 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 is 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.

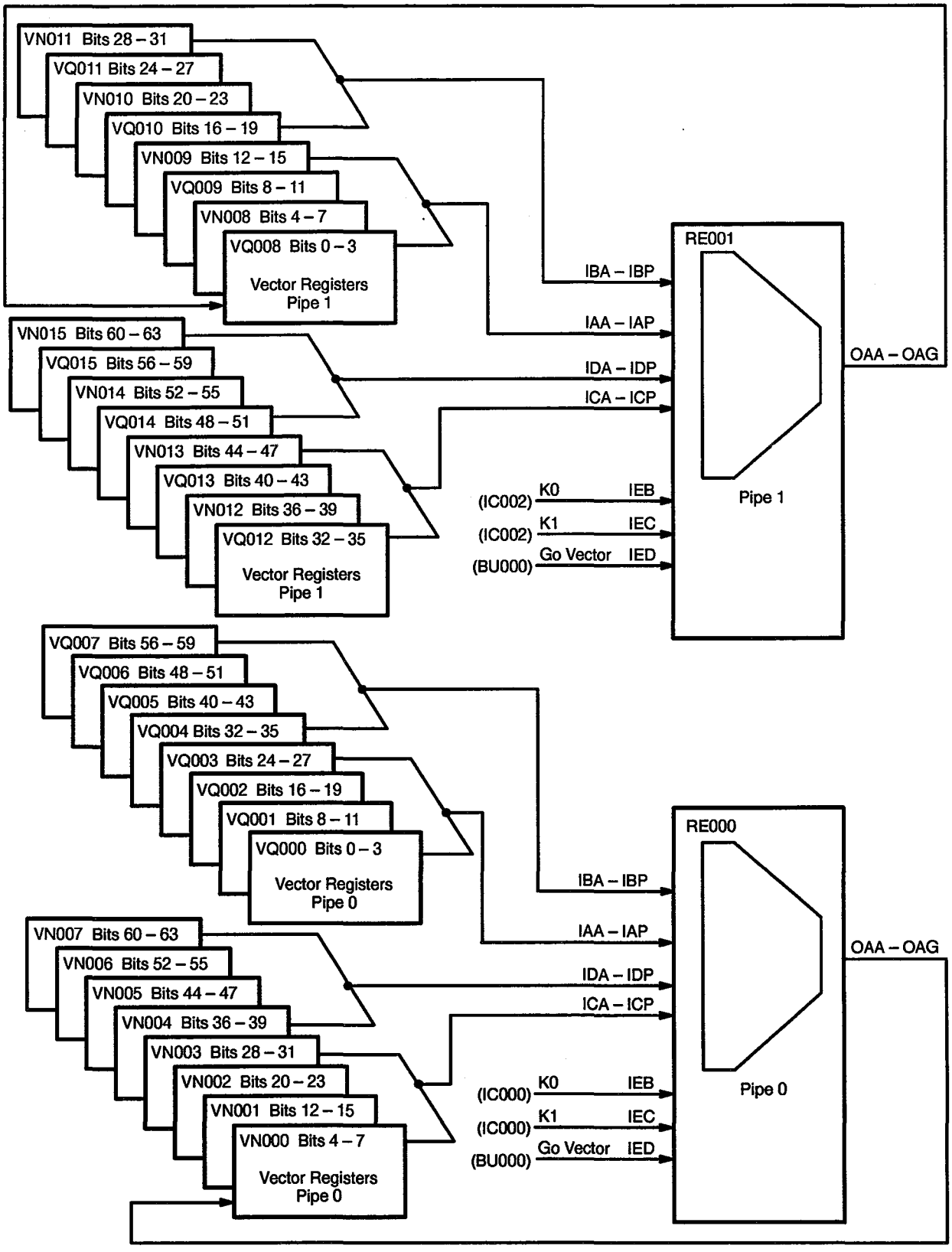


Figure 62. Vector Population/Parity/Leading Zero Block Diagram

Pop/Parity/Leading Zero Functional Units

The RE options contain part of the divide/square root unit and the logic for vector pop, vector pop parity, and vector leading zero. There are two RE options for each CPU. RE000 handles pipe 0 (the even elements), and RE001 handles pipe 1 (the odd elements).

The RE options receive data from the VN and VQ options; 4 bits come from each VQ and VN. Pop/parity/leading zero data uses the same wires and terms as the divide/square root data. The data is then sent to VN000 and VN008 on the same terms that the divide/square root 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 174*ij*1

Vector pop counts the number of bits set in a vector 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 174*ij*2

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 174*ij*3

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
174*ij*1	V*i* PV*j*	Population count (V*j*) to V*i*
174*ij*2	V*i* QV*j*	Parity of (V*j*) to V*i*
175*ij*3	V*i* ZV*j*	Transmit leading zero count of (V*j*) to V*i*

GATHER/SCATTER INSTRUCTIONS

The 176*i*1*k* and 1771*jk* instructions transfer blocks of data between common memory and the vector registers. The 176 instruction invokes the gather, or read function; the 177 instruction invokes the scatter, or write function. When the 176*i*1*k* 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 176*i*1*k* instruction transfers data from common memory to the V*i* register. Register A0 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 (A0) and the corresponding element of register V*k*. For example, during a 176213 instruction, V2[0] is loaded from address (A0) + (V3[0]); V2[1] is loaded from address (A0) + (V3[1]); etc.

The 005400 176*ijk* 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 A0 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 (A0) and the corresponding element of V*k*. For example, during a 005400 176213 instruction, V2[0] is loaded from address (A0) + (V3[0]); V2[1] is loaded from address (A0) + (V3[1]); etc. Simultaneously, V1[0] is loaded from address (A3) + (V3[0]); V1[1] is loaded from address (A3) + (V3[1]); etc.

Scatter Instructions

The 1771*jk* instruction transfers data from V*j* to common memory. The A0 register contains the initial address. V*k* contains the address indices.

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

IEEE FLOATING-POINT OVERVIEW

In general, the CRAY T90 series system CPE1 module conforms to the IEEE standard for binary floating-point arithmetic. It performs 64-bit floating-point add, subtract, multiply, divide and square root calculations. The CPE1 module also provides several new instructions that compare and convert floating-point and integer numbers.

The number and distribution of bits in the coefficient and the exponent (refer to Figure 63) are different than they are in the Cray proprietary floating-point format. (In the Cray proprietary floating-point format, the coefficient comprises bit 0 through bit 47; the exponent comprises bit 48 through bit 63.) Moreover, to ensure that the IEEE arithmetic results provide additional precision, bit -1 through bit -10 are appended in the logic to the right of the least significant bit of the coefficient. These supplemental bits are known as the Guard bit and the Sticky bits.

IEEE floating-point numbers are always represented as fractions – a number such as .1xxxxxxx...x raised to a power. The first bit in the fraction (the 1 bit, also called "the hidden bit") is always present in the hardware. Therefore all numbers in this computer system are considered *normalized numbers*; it is impossible to submit a number to the system that is not normalized. This bit, although invisible to the user, is included in the calculations. Consequently, calculations are made on a 53-bit fraction. The result that the user sees is in the form illustrated in Figure 63.

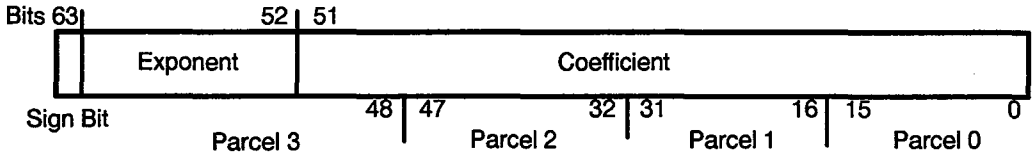


Figure 63. IEEE Floating-point Format

The benefits of the IEEE format are:

- Greater precision with 4 more bits in the coefficient field
- Specific representation for infinity and non-numeric numbers
- Control of rounding mode
- Consistency in handling of end-cases
- Expanded exceptions

IEEE Floating-point Number Examples

Table 23 lists some examples of IEEE floating-point numbers.

Table 23. IEEE Floating-point Numbers

Value	64-bit Word
+0	0000000000000000000
–0	1000000000000000000
+Greatest number	07775777777777777777
+Smallest number	00002000000000000000
Infinity	07776000000000000000
qNaN	07777xxxxxxxxxxxxxxx1
sNaN	07776xxxxxxxxxxxxxxx1

IEEE Terms

The following new terms are associated with IEEE floating-point:

Normal 0. Defined as an exponent of all 0's. The sign of a normal 0 may be positive or negative.

Denormalized. Defined as a minimum exponent in which the leading bit of the coefficient is equal to 0. *The CRI implementation of IEEE does not support denormalized numbers.* A denormalized number input into a floating-point unit will be converted to a zero before it is used. This is a departure from the IEEE standard.

Unnormalized. Defined as an unnormalized number in which the value of the exponent is greater than the minimum value of the format being used, and the leftmost bit of the significand is 0 (this number represents an unnormalized 0). *Only normalized number representations are supported.*

Normalized. Defined as a nonzero number in which the leftmost bit of the significand is a 1. If the significand is a 0 then the number becomes a normal 0. Normalization does not change the sign of the number.

NaN. Defined as a symbolic entity encoded in floating-point format and resulting from an operation that has no mathematical interpretation. For example, 0 divided by 0 produces a NaN (Not A Number).

Rules of Operation for NaNs

- The sign of a NaN is never significant.
- When any floating-point unit receives a NaN, it generates an Invalid (NVI) signal and returns a result of NaN.
- There are two different types of NaNs: quiet and signaling. If the most significant bit of the coefficient is a 1, the NaN is considered quiet. When a single operand NaN is received by the floating add or floating multiply unit, that NaN is returned as a result except that:
  - A signaling NaN is converted to a quiet NaN.
  - The sign is converted to positive.
- When two signaling NaNs or two quiet NaNs are received by the floating add or floating multiply unit, the *j* operand is returned as a result and modified in compliance with the single-operand NaN rule.
- When a signaling NaN and a quiet NaN are received by the floating add or floating multiply unit, the signaling NaN is returned as a result and modified according to the preceding rule.

- When either the divide or square root unit receives a NaN, it returns a quiet NaN with all bits set in the coefficient.
- NaNs that are generated within a floating-point unit that were not caused by receiving a NaN as an operand are given a tag code, which is returned as part of the result. The result returned will be all bits set except for bits 48, 49, and 50. These bits will show which functional unit generated that result. Table 24 lists the NaN tag codes.

Table 24. NaN Tag Codes

Functional Unit	Bit 50	Bit 49	Bit 48
Add	0	0	1
Multiply	0	1	0
Divide	1	0	0
Square Root	1	0	1

- When NaNs are sent to the compare unit:
  - NaNs never compare to another operand
  - NaNs never compare to another NaN
  - NaNs are not equal to another NaN
  - NaNs always fail equality tests and pass inequality tests
  - The unordered test returns true if either input is a NaN

Deviations from the IEEE Standard

In the following cases, CRI does not follow the IEEE standard:

- Only 64-bit format, no support for the 32-bit format
- No support for denormalized numbers
- Exception flags are not precise because of a lack of instruction ordering

Special Operand Values

Three special operand cases that are considered in IEEE are as follows:

- Any floating-point operand with an exponent field of all 0's is considered a zero value. The sign is significant.
  - *+nnn* x –0 = –0
  - –n x +0 = –0
  - *+nnn* – *+nnn* = +0 (except if rounding down)
  - Sqrt –0 = –0
  - +0 result rounded down = –0
  - Compare instructions +0 = –0

- When there is a maximum exponent and the coefficient is all 0's, the operand is considered to be infinite. The sign is significant. Infinite values are generated when the exponent range required to represent the number is exceeded. The value is operated on and exceptional results are generated (overflow).
  - 0777600000000000000000 = positive infinity
  - 1777600000000000000000 = negative infinity

- When there is a maximum exponent and the coefficient is *not* all 0's, the operand is not considered to be a real number (NaN). The sign is ignored. There are two different types of NaNs: quiet qNaN and signaling sNaN. If the most significant bit of the coefficient is a 1, the NaN is considered quiet. A qNaN is operated on like all other operands; however, an exceptional input exception signal is generated in the status register. If an sNaN is received as an operand, an invalid signal is generated.
  - 077760xxxxxxxxxxxxxxxxx1 = Quiet NaN (qNaN)
  - 077770xxxxxxxxxxxxxxxxx1 = Signaling NaN (sNaN)

Floating-point Exception (Flags)

Floating-point operations can generate several exception flags. These exceptions can be seen in Status register SR0. Associated with these exceptions are interrupt bits. The interrupt bits can be enabled or disabled by the user. An interrupt will be generated if the exception is enabled, and then a status register bit is set. *If an exception is set and then the user enables the interrupt, no interrupt will be generated.* This is different from previous Cray computer systems.

For instructions that can change interrupt mode bits, floating-point instruction issue halts until all floating-point functional units are quiet. All floating-point operations will complete with the same interrupt modes that were set when they began.

There are six exceptions; they are:

- **Invalid (NVI)** An attempt was made to generate a result that is not a real number. Invalid is signaled for the following reasons:
  - A signaling NaN (sNaN) was received as an input operand
  - Addition or subtraction of infinite operands in some cases
  - +∞ – +∞ = invalid
  - Multiplication of 0 x infinity
  - Division of 0 / 0 or infinity / infinity
  - Square root of any negative number
  - Signed compare where one or both inputs are NaNs (>, <=, <, >=) Every NaN shall compare unordered

- **Divide by 0 (DVI)** An attempt has been made to divide a finite normal numerator by zero.

- **Overflow (OVF)** A result larger than the greatest representable number was generated. A positive infinity is returned (07776000000000000000). Overflow is handled differently than the IEEE standard. Overflow is carried to positive or negative infinity when rounding away from zero, and to the largest finite number when rounding toward zero when the interrupt on overflow is disabled. The standard specifies that when interrupt on overflow is enabled, the operation will deliver the result, with the exponent biased toward zero by 3000_8. Cray Research floating-point units cannot detect whether the interrupts are enabled or disabled, and therefore are unable to handle the two cases differently.

- **Underflow (UNF)** A result smaller than the least representable number was generated. A coefficient of zero with the sign bit is returned. (00000000000000000000). This result is different from the IEEE standard. The IEEE standard returns the result obtained after multiplying the infinitely precise result by 2^{α} (where 2^{α} is the bias adjust) and then rounding.

- **Inexact (NX)** A result was generated that would be different if all possible significant bits were returned or could be returned. Inexact is also signaled on both overflow and underflow when the returned result is not exactly zero. For example, 1 divided by 3 returns the repeating decimal, 0.33333.....3, and signals Inexact.

- **Exceptional Input (XI)** A floating-point unit received an operand of infinity or NaN. XI is a Cray feature, not an IEEE standard.

Rounding

Rounding is done by adding 1 to the least significant bit (LSB) of the result if it is determined to be required by the rounding mode bits and any bit of less significance than the LSB of the coefficient.

The first bit to the right of the LSB is called the guard bit; all the bits to the right of the guard bit are "ORed" together into a "sticky" bit. If the guard bit and the sticky bits are all 0's, then the results are exact and no rounding occurs. If either bit is a 1, then inexact is signaled and a 1 is added to the LSB, depending on the rounding mode.

There are four rounding modes that apply to the floating-point units:

- **Round to the nearest.** The result closest to infinitely precise is returned. If the bits to the right of the LSB are greater than half the value of the LSB, a 1 is added to the results. If the bits to the right of the LSB are exactly half the value of the LSB, a 1 is added to the results if the LSB=1.
- **Round up.** The more positive result closest to infinitely precise is returned.
- **Round to zero.** The result closest to zero is returned.
- **Round down.** The more negative result is returned.

Table 25 shows the effect of the sign bit, guard bit and sticky bit on the LSB, depending on the rounding mode selected.

Table 25. Effects of Rounding Mode on LSB

Result Bits			Rounding Mode			
Sign Bit	Guard Bit	Sticky Bit	Round to Nearest	Round to Zero	Round Up	Round Down
x	0	0	No	No	No	No
0	0	1	No	No	Yes	No
0	1	0	Yes if LSB=1	No	Yes	No
0	1	1	Yes	No	Yes	No
1	0	1	No	No	No	Yes
1	1	0	Yes if LSB=1	No	No	Yes
1	1	1	Yes	No	No	Yes

IEEE Mathematical Functions

With the inclusion of NaN and infinity operands, more exceptional results are possible. Table 26 through Table 28 show the results from different combinations of operands and different operations. Remember to consider the state of the rounding mode when you calculate the final results.

Addition and Subtraction Rules

Addition of equal operands with opposite signs produces a zero result. A positive zero results if rounding mode is set to *round to nearest* or *round up* or *round to zero*. A negative zero results if the *round down* mode is used. A zero value is also returned if the operation underflows; the sign of the result is the sign determined before underflow occurs. If the operation signals overflow and the rounding mode is set to *round to nearest* or *round up*, the result returned is a +∞ (0777600000000000000000). If the round mode is set to *round to zero* or *round down*, the result rounds to the greatest representable value (0777577777777777777777).

Table 26. Addition and Subtraction Results

k operand	*j* operand			
	n	0	∞	NaN
n	0, n, ∞	n	∞	NaN
0	n	0	∞	NaN
∞	∞	∞	∞, NaN*	NaN
NaN	NaN	NaN	NaN	NaN

* A NaN is returned when adding two ∞ of different signs.
Subtracting two ∞ of different signs results in a result of ∞ with the sign of the minuend.

Multiplication, Division, and Square Root Rules

Multiplication or division of two nonzero numbers results in zero only if the operation detects underflow. If an overflow occurs, a ±∞ (0777600000000000000000) or the greatest representable value (0777577777777777777777) is returned, depending on the rounding mode.

Table 27. Multiplication Results

k operand	*j* operand			
	n	0	∞	NaN
n	0, n, ∞	0	∞	NaN
0	0	0	NaN	NaN
∞	∞	NaN	∞	NaN
NaN	NaN	NaN	NaN	NaN

Table 28. Division Results

k operand	*j* operand			
	n	0	∞	NaN
n	0, n, ∞	0	∞	NaN
0	∞	NaN	∞	NaN
∞	0	0	NaN	NaN
NaN	NaN	NaN	NaN	NaN

Table 29. Square Root Results

j operand	+n	±0	−n	NaN
Results	+n	±0	NaN	NaN

IEEE FLOATING-POINT ADD AND COMPARE

The floating-point add unit is contained on the FC options. The FC options perform the following four types of operations:

- IEEE floating add and subtract
- IEEE floating point-to-integer conversion
- IEEE integer-to-floating point conversion
- IEEE compare instructions

There are three FC options in each CPU. Each FC option has a specific function.

- FC000
    - Performs all scalar-to-scalar floating add functions
    - Performs all scalar-to-scalar compare functions
    - Performs all scalar-to-scalar conversions
    - Passes all pipe 0 vector data
- FC001
    - Performs all pipe 0 floating add functions
    - Performs all scalar-to-vector (S*j* V*k*) compare functions
    - Performs all vector-to-vector (V*j* V*k*) compare functions
    - Performs all vector-to-vector conversions for pipe 0
    - Passes all output data to FC000
- FC002
    - Performs all pipe 1 floating add functions
    - Performs all scalar-to-vector (S*j* V*k*) compare functions
    - Performs all vector-to-vector (V*j* V*k*) compare functions
    - Performs all vector-to-vector conversions for pipe 1

Floating Point Addition / Subtraction

The floating add functional unit, like the floating-point multiply unit, receives normalized numbers as inputs. Because of the hidden bit, all numbers are normalized. An input number that contains an exponent of 0's will clear the coefficient to 0 before using it as an operand in the functional unit. NaN operands are handled in accordance with the IEEE standard. Performing an add or subtract operation on a NaN results in a NaN being produced and a flag set.

Four IEEE standard flags and one non-IEEE standard flag are used in the floating-point add unit. They are:

- **Invalid (NVI)** An attempt was made to generate a result that is not a number. NVI is signaled for the following conditions:
  - A NaN as an input operand
  - Addition or subtraction of infinity
  - Signed compare with at least one NaN input
  - Attempt to convert an out-of-range number
- **Overflow (OVF)** A result larger than the greatest representable number has been generated. Positive infinity (07776000000000000000) is returned. The CRAY T90 series version of IEEE treats OVF differently than the IEEE standard. In the CRAY T90 series application, overflow is carried to positive or negative infinity when rounding away from zero. Overflow is carried to the largest finite number when rounding towards zero, when the interrupt on overflow is disabled. The IEEE standard specifies that the operation will deliver the result, with the exponent biased toward zero by 3000 when interrupt on overflow is enabled. The floating-point units have no way to detect whether the traps are enabled or disabled, and therefore are unable to handle the two cases differently.
- **Underflow (UNF)** A result smaller then the least representable number was generated. A value of zero with the sign bit ±(00000000000000000000) is returned.
- **Inexact (NX)** A result was generated whose value would be different if all possible significant bits were returned or could be returned. Inexact is also signaled on both overflow and underflow when the result is not exactly 0. Some examples of inexact numbers are repeating decimals and pi.

- **Exceptional Input (XI)** A floating-point unit received either an infinite or NaN operand. XI is a CRI feature that is not an IEEE standard.

Figure 64 is a diagram of the floating add functional unit. The functional unit uses 2 round mode bits to select one of four rounding modes. Table 30 shows the four rounding modes used by the FC options.

Table 30. Rounding Modes

Round Mode	(RM0)	(RM1)
Nearest	0	0
Up infinity	1	0
To zero	0	1
Down infinity	1	1

You can set the rounding modes either by issuing an instruction or by setting a bit in the exchange package. The 003004 through 003007 instructions set the rounding mode directly; the 005400 073*i*05 instruction sets the rounding mode from the contents of S*i*. A change to the rounding mode affects all floating-point instructions issued thereafter, but it has no effect on instructions issued previously. The two exchange package bits, RM0 and RM1, determine the rounding modes (as illustrated in Table 30).

Rounding is determined by the choice of rounding mode and the values of the guard bit, the sign bit, the sticky bits, and the least significant bit (LSB). Table 31 defines when a 1 bit is added to the LSB of the results.

Table 31. Effects of Rounding Mode on LSB

Result Bits			Rounding Mode			
Sign Bit	Guard Bit	Sticky Bit	Round to Nearest	Round to Zero	Round Up	Round Down
x	0	0	No	No	No	No
0	0	1	No	No	Yes	No
0	1	0	Yes if LSB=1	No	Yes	No
0	1	1	Yes	No	Yes	No
1	0	1	No	No	No	Yes
1	1	0	Yes if LSB=1	No	No	Yes
1	1	1	Yes	No	No	Yes

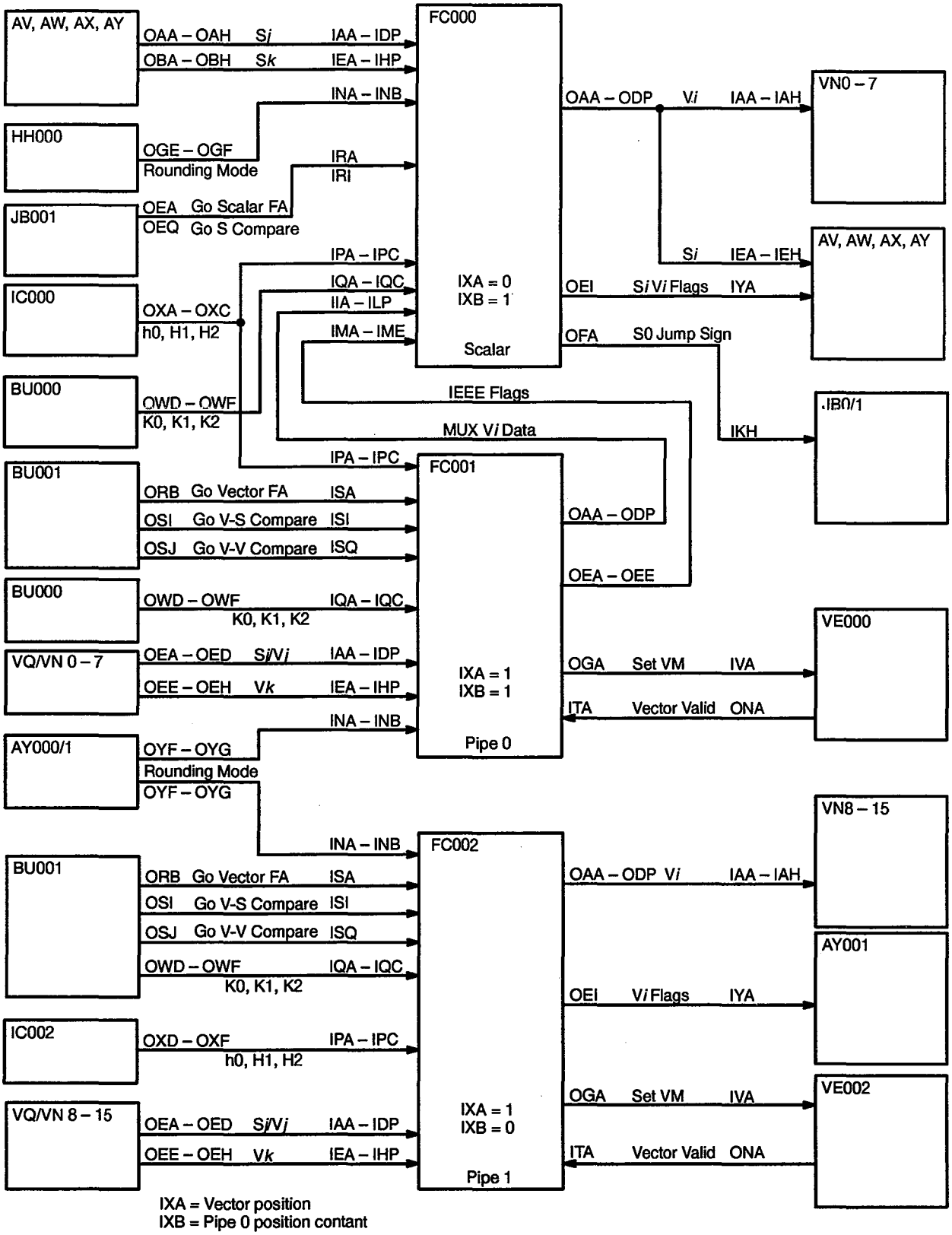


Figure 64. Floating Add Functional Unit

Floating-point Add Functional Unit Instructions

Refer to Table 32 for a list of the floating-point add functional unit instructions.

Table 32. Floating-point Add Functional Unit Instructions

Instruction	CAL	Description
062*ijk*	S*i* S*j* + FS*k*	Scalar floating-point sum of (S*j*) and (S*k*) to S*i*
063*ijk*	S*i* S*j* – FS*k*	Scalar floating-point difference of (S*j*) minus (S*k*) to S*i*
170*ijk*	V*i* S*j* + FV*k*	Vector floating-point sum of (S*j*) and (V*k* elements) to V*i*
171*ijk*	V*i* V*j* + FV*k*	Vector floating-point sum of (V*i* elements) and (V*k* elements) to V*i*
172*ijk*	V*i* S*j* – FV*k*	Vector floating-point difference of (S*j*) minus (V*k*) to V*i*
173*ijk*	V*i* V*j* – FV*k*	Vector floating-point difference of (V*j* elements) minus (V*k* elements) to V*i*

Floating-point Format

Refer to Figure 65 for an illustration of floating-point format. Consider a floating-point number *normalized* when the most significant bit of the coefficient (bit 51) is set.

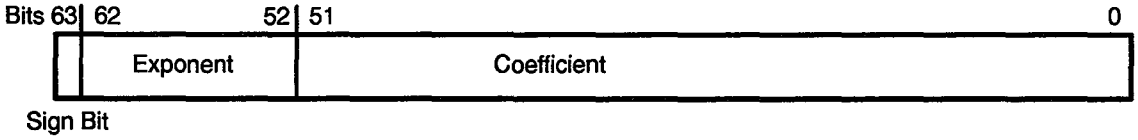


Figure 65. IEEE Floating-point Format

Floating-point-to-Integer Conversion

Floating-point-to-integer conversion takes place on the FC options. This operation converts a floating-point number to a signed 64-bit integer. There are two cases of this conversion instruction. One case converts without rounding and is not IEEE standard. The other case enables rounding. Table 33 describes the floating-point-to-integer conversion instructions.

Table 33. Floating-Point-to-Integer Conversion Instructions

Instruction	CAL	Description
070*ij*2	S*i* int, S*j*	Floating-point S*j* to integer S*i*
070*ij*3	S*i* rint, S*j*	Floating-point S*j* to rounded integer S*i*
167*ij*0	V*i* int, V*j*	Floating-point V*j* to integer V*i*
167*ij*1	V*i* rint, V*j*	Floating-point V*j* to rounded integer V*i*

There are some notable special cases that involve the instructions listed in Table 33. The invalid signal is sent:

- If the *j* field of the instruction is a 0, then (S*j*) or (V*j*) is 0. The result is +0 (0000000000000000000).
- If the floating-point number has a value greater than $2^{64}-1$, then the unit will return $\pm$0777577777777777777777. This value is the largest number that can be represented.
- If the input is a NaN, then +0777777777777777777777 is returned and **invalid** is signaled.
- If the input value is less than 1, a 0 or a 1 is returned, depending on the rounding mode. The **inexact** signal will be sent unless the input operand was exactly 0.

Integer-to-Floating-Point Conversion

Integer-to-floating-point conversions occur on the FC options. Two instructions can convert a signed 64 bit integer into a floating-point number. The result will be exact if the absolute value of the source operand is less than 2^{54}. Otherwise the result is rounded, using the current rounding mode. Refer to Table 34 for a description of the two integer-to-floating-point conversion instructions.

Table 34. Conversion Instructions

Instruction	CAL	Description
070*ij*4	S*i* flt, S*j*	Integer S*j* to floating-point S*i*
167*ij*2	V*i* flt, V*j*	Integer V*j* to floating-point V*i*

Floating-point Comparisons

The IEEE standard supports a full set of floating-point comparison instructions. There are four mutually exclusive operations that are possible, they are:

- Less than
- Greater than
- Equal
- Unordered

Comparisons are always exact. They never overflow, underflow, or signal inexact exceptions. If a signaling NaN (bit 51 of the fraction is 0) is received as an input, it will generate an exception (XI) interrupt and also an invalid (NVI) interrupt for signed compare tests (>, >=, <, <=). An invalid also occurs if a quiet NaN (bit 51 of the fraction is 1) is received in a signed compare test (>, >=, <, <=). Note that a NaN will always fail an equal test (NaNs are equal to nothing) and always pass the Not equal test.

For compare functions, the sign of a zero value is ignored. Therefore a positive zero will equal a negative zero, and a positive zero is not greater than a negative zero.

When a scalar compare instruction tests true for a condition, all of the bits in the result register are set. If the test fails, the result register will contain 0's. For vector operations, passing a test sets a bit in the mask register and failing a test clears the corresponding bit in the mask register. Table 35 lists the instructions used in the compare function.

Table 35. Compare Instructions

Instruction	CAL	Description
005501 164*ijk*	S*i* S*j*,EQ,S*k*	Floating-point compare equal
005502 164*ijk*	S*i* S*j*,NQ,S*k*	Floating-point compare not equal
005503 164*ijk*	S*i* S*j*,GT,S*k*	Floating-point compare greater than
005504 164*ijk*	S*i* S*j*,LE,S*k*	Floating-point compare less than or equal
005505 164*ijk*	S*i* S*j*,LT,S*k*	Floating-point compare less than
005506 164*ijk*	S*i* S*j*,GE,S*k*	Floating-point compare greater than or equal
005507 164*ijk*	S*i* S*j*,UN,S*k*	Floating-point compare unordered
005521 1640*jk*	VM S*j*,EQ,V*k*	Floating-point compare equal
005522 1640*jk*	VM S*j*,NQ,V*k*	Floating-point compare not equal

Table 35. Compare Instructions (continued)

Instruction	CAL	Description
005523 1640*jk*	VM S*j*,GT,V*k*	Floating-point compare greater than
005524 1640*jk*	VM S*j*,LE,V*k*	Floating-point compare less than or equal
005525 1640*jk*	VM S*j*,LT,V*k*	Floating-point compare less than
005526 1640*jk*	VM S*j*,GE,V*k*	Floating-point compare greater than or equal
005527 1640*jk*	VM S*j*,UN,V*k*	Floating-point compare unordered
005541 1640*jk*	VM V*j*,EQ,V*k*	Floating-point compare equal
005542 1640*jk*	VM V*j*,NQ,V*k*	Floating-point compare not equal
005543 1640*jk*	VM V*j*,GT,V*k*	Floating-point compare greater than
005544 1640*jk*	VM V*j*,LE,V*k*	Floating-point compare less than or equal
005545 1640*jk*	VM V*j*,LT,V*k*	Floating-point compare less than
005546 1640*jk*	VM V*j*,GE,V*k*	Floating-point compare greater than
005547 1640*jk*	VM V*j*,UN,V*k*	Floating-point compare unordered

IEEE DIVIDE AND SQUARE ROOT

IEEE Divide

The vector and scalar registers share the divide and square root functional unit. The divide functional unit also handles the iota instructions and the pop, parity, and leading zero operations. (These functions are discussed in the Vector Logical and in the Vector Pop/Parity sections.) There are two divide and square root pipes; each pipe consists of one RE option and two RD options. (Refer to Figure 67 at the end of this section for a block diagram of the divide functional unit.)

All input data from the vector and scalar registers arrives at the functional unit from the vector options. Scalar data is also routed through the vector options, using the same path to the RE options.

NOTE: The divide unit operates in either full- or half-precision mode. Although the hardware for half-precision is on the module, there is no compiler or software support for the half-precision instructions.

In half-precision mode, the divide unit stops iterating after 16 iterations and produces 32-bit results. In full-precision mode, the divide unit performs 28 iterations. The top bit of the result is generally a 0, but it can be 1 if the ratio of the mantissa to the radicand is approximately 2:1. The next bit is the hidden bit if no left shift is required. The hidden bit is 2 bits below the top bit if a left shift is required, which leaves 29 or 30 bits to the right of the hidden bit. The remaining (unused) 22 or 23 bits are set to 0's.

Table 36 lists the IEEE floating-point divide and square root instructions that are available on CRAY T90 series systems.

Table 36. Floating-point Divide and Square Root Unit Instructions

Instruction	CAL	Description
065*ijk*	S*i* S*k*/FS*j*	Floating-point S*k* divided by S*j* to S*i*.
065*ijk* *	S*i* S*k*/HS*j*	Half precision floating-point S*k* divided by S*j* to S*i*.
070*ij*0	S*i* SQR S*j*	Floating-point square root of S*j* to S*i*.
070*ij*0*	S*i* SQRH S*j*	Half precision floating-point square root of S*j* to S*i*.
162*ijk*	V*i* V*k*/FS*j*	Floating-point V*k* divided by S*j* to V*i*.
162*ijk**	V*i* V*k*/HS*j*	Half precision floating-point V*k* divided by S*j* to V*i*.
163*ijk*	V*i* V*k*/FV*j*	Floating-point V*k* divided by V*j* to V*i*.
163*ijk**	V*i* V*k*/HV*j*	Half precision floating-point V*k* divided by V*j* to V*i*.
174*ijk*0	V*i* SQR V*j*	Floating-point square root of V*j* to V*i*.
174*ijk*0*	V*i* SQRH V*j*	Half precision floating-point square root of V*j* to V*i*.

* Must be preceded by a 005400 instruction

Divide/Square Root Options

There are two sets of options because this functional unit has two pipes. The even elements are processed by pipe 0, and the odd elements are processed by pipe 1. Table 37 shows the options used for each pipe.

Table 37. Divide Options

Pipe 0	Pipe 1
RE000	RE001
RD000	RD002
RD001	RD003

RD option

The RD option communicates only with the RE option; there are two RD options for each RE option. The RD receives input operands from the RE option: first the *j* operand, then the *k* operand. The RD option sends the mantissa serially to the RE option.

Each RD option contains four identical divide/square root cores. There are a total of eight cores in each unit. Divide and square root operands are sent to the RD options so that each RD option receives operands at a maximum rate of one every 4 clock periods (CPs) in half-precision mode or one every 8 CPs in full-precision mode. Operands are always sent to the even-numbered RD option first. If a new divide operation is starting and it has been at least 16 CPs since the last operation, the unit will reset the pipe back to the even-numbered RD option. This feature allows a failure to be isolated to a particular RD option.

The input data is received at the RE option and sent to the RD option along with the Yugo signal. The Yugo signal causes the RD option to assign one of the divide cores to begin calculation.

RE Option

There is one RE option for each pipe. The RE option is responsible for:

- Iota (See the "Vector Logical" section for a description of Iota)
- Vector Pop/Pop Parity and Leading Zero instructions (See the "Vector Pop/Pop Parity and Leading Zero" section for a description of these instructions)
- Exponents calculation
- Exceptions
- Normalization
- Rounding

All communication with the CPU occurs through the RE options. There is only one 64-bit operand path into the divide unit. The divide unit receives data from the VN and VQ options and passes it on to the RD option. The *j* and *k* operands for divide are multiplexed; first *j* arrives and then *k*. Scalar data is also routed through the VN and VQ options.

Normalization

A floating-point divide operation may be normalized at most by one position. If the divisor is greater then the mantissa, then the most significant bit of the result is 0 and a left shift of one position is preformed. Otherwise, the most significant bit of the result is always a 1.

Square root operations should never require normalization. The radicand is shifted left one position before the operation is started. There is one exception. Although it is mathematically impossible for the ratio of two mantissas to be equal to 2, or the square root of n<4 to be 2, it is possible, in half-precision mode, for this result to be produced. Also if rounding away from zero, the square root of the largest possible n<4 must be rounded up to 2. In all these cases, the bit above the most significant bit is set and all other bits are forced to 0's. For square root, this case is detected and the exponent is adjusted accordingly. For divide, the exponent is left unjustified and the mantissa is forced to 0.

Rounding

Two rounding mode bits are received at the RE option and held for vector length. The 2 rounding mode bits select one of following four rounding modes:

- 00 = Round to nearest
- 01 = Round toward positive
- 10 = Round toward zero
- 11 = Round toward negative

Rounding occurs by adding one to the least significant bit (LSB) of the results. (Rounding is determined to be required by the rounding mode bits and any bit of less significance than the LSB of the coefficient and possibly the sign bit and the LSB.)

In rounding, the first bit to the right of the LSB is called the guard bit, all the bits to the right of the guard bit are "ORed" together into a "sticky" bit. If the guard bit and the sticky bit are 0's, then the results are exact and no rounding will take place. If either bit is a 1, then Inexact is signaled and a 1 is added to the LSB, depending on the rounding mode.

Floating Point Exception Flags

The divide square root unit has six exception flags:

- **Invalid (NVI)** An attempt has been made to generate a result that is not a real number. Invalid is signaled for the following conditions:
  - A signaling NaN (sNaN) was received as an input operand
  - Division of 0 by 0 or infinity by infinity
  - Square root of a negative number

- **Divide by 0 (DVI)** An attempt has been made to divide a finite normal numerator by zero.
- **Overflow (OVF)** A result that is larger than the largest representable number was generated.
- **Underflow (UNF)** A nonzero result that is smaller than the smallest representable number was generated.
- **Inexact (NX)** A result was generated whose value would be different if all possible significant bits were returned or could be returned. Inexact is also signaled on both overflow and underflow when the result is not exactly 0. For example, 1 divided by 3 returns the repeating decimal, 0.33333.....3, and signals Inexact.
- **Exceptional Input (XI)** A floating-point unit received an operand of infinity or NaN. XI is a CRI feature, not an IEEE standard.

Exception flags and other generated information about the operation are sent serially to the AY option and onward to the status registers of the HH options. The information is recoded and staged as shown in Figure 66.

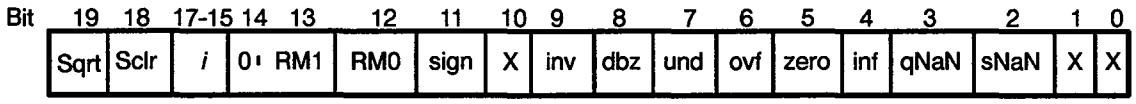


Figure 66. Serial Floating-point Status

Division and Square Root Rules

If either operand of a divide is a NaN, or if the operand in a square root is a NaN, or if the operation is invalid, then the result must be a NaN. If one of the operands is a NaN, the result will be a positive value quiet NaN, with a mantissa of all 1's. If a NaN is generated because of an invalid operation, the result will be a positive value quiet NaN, but 1 or 2 bits of the mantissa will be set to identify which unit generated the NaN. These identifier bits are shown in Table 38.

Table 38. NaN Identifiers

Unit	Bit 50	Bit 49	Bit 48
Divide	1	0	0
Square Root	1	0	1

Division of two nonzero numbers results in a 0, only when an underflow operation occurs. If overflow occurs, a $\pm\infty$ (0777600000000000000000) or the greatest representable value (0777577777777777777777) is returned. Table 39 lists the characteristics of floating divide input operands and how they affect the quotient. Table 40 contains a list for square root calculations.

Table 39. Division Results

k operand	*j* operand			
	n	0	∞	NaN
n	0, n, ∞	0	∞	NaN
0	∞	NaN	∞	NaN
∞	0	0	NaN	NaN
NaN	NaN	NaN	NaN	NaN

Table 40. Square Root Results

j operand	+n	$\pm$0	–n	NaN
Results	+n	$\pm$0	NaN	NaN

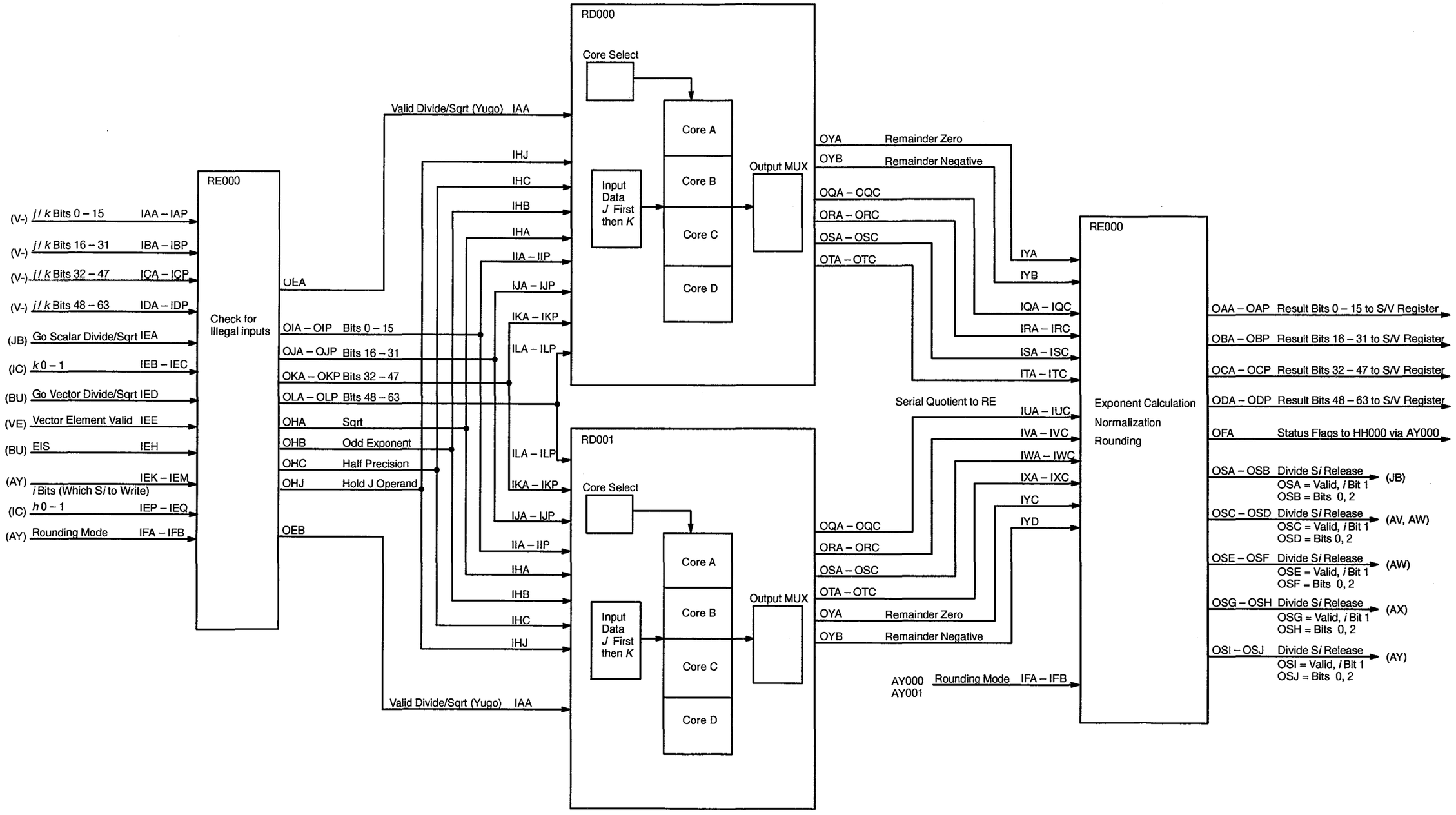


Figure 67. Divide Unit Block Diagram

IEEE FLOATING-POINT MULTIPLY AND INTEGER MULTIPLY

The scalar and vector registers share the floating-point multiply functional unit. Two floating-point operands arrive at the multiply functional unit from 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 51-bit coefficients are multiplied.

The floating-point multiply functional unit also performs the integer multiply operation. Two 64-bit operands arrive at the functional unit and a 128-bit result is generated. With the EIS instruction set, the user can select either the upper 64 bits or the lower 64 bits of the result.

The multiply unit is a dual pipe unit. Each unit consists of five options: the NE option, two NF options, an HG option, and an NH option. Refer to the block diagrams of the multiply functional unit in Figure 73 and Figure 74.

Multiply Algorithm

The multiply functional unit uses a type of recode multiplication algorithm known as Booth's Algorithm.

The multiplier, in this case the *j* operand, is partitioned into 3-bit recode groups centered on the even bits. A forced zero is added to the first recode group. The recode groups are formed as shown in Table 41. The following subsections provide examples of standard and Booth Recode multiplication.

Table 41. Recode Groups

Odd Bit	Even Bit	*i*–1	Recode Value	Recode Product
0	0	0	+0	0
0	0	1	+1	X
0	1	0	+1	X
0	1	1	+2	2X
1	0	0	–2	(2X)'+1
1	0	1	–1	(X)'+1
1	1	0	–1	(X)'+1
1	1	1	–0	(0)' +1
i – 1 = Bit to right of recode group		X = Multiplicand		

Standard Binary Multiplication

Refer to the following example of standard binary multiplication:

```
          000011  (3)
          011101  (35)
        ---------
          000011
        000000
       000011
      000011
     000011
000000
-------------------
0000001010111  (127)
```

Booth Recode Multiplication

Refer to the following example of Booth Recode multiplication:

```
                    000011  (3)
                    011101  (35)
              ---------------
                000000000011
                 11111111010
                 00000110
              ---------------
            1   000001010111  (127)
```

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 41, the first recode of the multiplier, bit 1, bit 0, and the forced zero, produces a recode value of 010, or +1. In this case, the multiplicand is brought down to form the first partial product.

The second recode, bit 3, bit 2, and bit 1, produces a recode value of –1. In this case, a two's complement and a shift of 1 are performed on the multiplicand, which forms the second partial product.

The final recode, bits 5, 4, and 3 produces a recode value of +2, which results in a shift of 1 on the multiplicand and forms the third partial product.

Integer Multiply Instructions

The floating-point multiply functional unit also performs the integer multiply operation. Two 64-bit operands are presented to the unit and a 128-bit result is generated. The EIS instruction set allows the user to select either the upper 64 bits or the lower 64 bits of the 128-bit result. Refer to Table 42 for a list of the integer multiply instructions.

Table 42. Integer Multiply Instructions

Instruction	CAL	Description
066*ijk*	S*i*S*j**LS*k*	Integer product, (S*j*) times (S*k*) to S*i*, returning lower
066*ijk* *	S*i*S*j**US*k*	Integer product, (S*j*) times (S*k*) to S*i*, returning upper
165*ijk*	V*i*V*j**LV*k*	Integer product, (V*j* elements) times (V*k* elements) to V*i*, returning lower
165*ijk* *	V*i*V*j**UV*k*	Integer product, (V*j* elements) times (V*k* elements) to V*i*, returning upper
166*ijk*	V*i*S*j**LV*k*	Integer product, (S*j*) times (V*k* elements) to V*i*, returning lower
166*ijk* *	V*i*S*j**UV*k*	Integer product, (S*j*) times (V*k* elements) to V*i*, returning upper

* Must be preceded by a 005400 instruction

Floating-point Multiply Instructions

The floating point-multiply unit uses the IEEE standard for multiplication. There are 11 exponent bits and 52 coefficient bits. Refer to Figure 68 for the IEEE format.

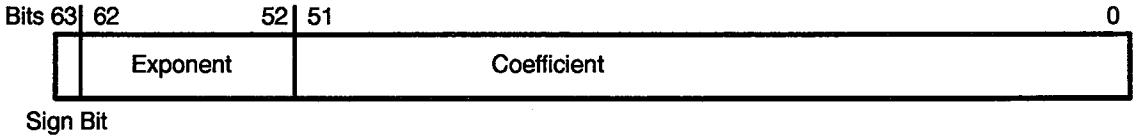


Figure 68. IEEE Floating-point Format

When two operands are presented to the unit, a pyramid is formed. The least significant bits are captured by the NE option (NE000 for pipe 0 and NE001 for pipe 1). These bits are the sticky bits when rounding modes are in operation, and they are also the lower bits of the integer multiply results. The two NF options, (NF000 and NF001 for pipe 0 and NF002 and NF003 for pipe 1) form the middle of the pyramid.

Refer to Table 43 for a list of the floating-point multiply instructions.

Table 43. Floating-point Multiply Instructions

Instruction	CAL	Description
064*ijk*	S*i*S*j**FS*k*	Scalar floating-point product of (S*j*) times (S*k*) to (S*i*)
160*ijk*	V*i*S*j**FV*k*	Vector floating-point product (S*j*) times (V*k* elements) to V*i*
161*ijk*	V*i*V*j**FV*k*	Vector floating-point product (V*j* elements) times (V*k* elements) to V*i*

Multiply Functional Unit Options

There are two sets of options because the multiply functional unit is a dual-pipe functional unit. The even elements are processed by pipe 0, and the odd elements are processed by pipe 1. Table 44 shows the options used for each pipe.

Table 44. Multiply Options

Pipe 0	Pipe 1
NE000	NE001
NF000	NF002
NF001	NF003
NG000	NG001
NH000	NH001

NE Option

The NE option forms the rightmost (least significant) portion of the pyramid. (Refer to Figure 69.) The NE option receives S*k* and V*k* operand bits 0 through 49 and S*j* and V*j* bits 0 through 50. During a floating multiply operation, this portion of the pyramid is used mainly to create the sticky bits; however, during an integer multiply, the results will be used to produce the full 128-bit result. The NE option receives very little control from the rest of the unit. It cannot distinguish whether the operands are to be used as floating point or integer.

NF Option

There are two NF options per pipe. NF000 and NF001 are used for pipe 0, and NF002 and NF003 are used for pipe 1. A particular input may be used on one option and not the other, depending on its position. The NF option may receive control signals from the NG option with information that an invalid operand was received and instructions to abort further calculations.

NF000 receives S*k* and V*k* operands bit 33 through bit 65 (bit 64 and bit 65 are forced to 0's) and S*j* and V*j* bit –1 through bit 47 (bit –1 is forced to a zero). NF000 generates the upper-middle portion of the pyramid. (Refer to Figure 70).

NF001 receives S*k* and V*k* operand bit –1 through bit 33, (bit –1 is forced to zero) and S*j* and V*j* bit 17 through bit 65 (bit 64 and bit 65 are forced to 0's). NF001 generates the lower-middle portion of the pyramid. (Refer to Figure 71.)

NG Option

The NG option forms the left portion of the pyramid. (Refer to Figure 72.) The NG option receives S*k* and V*k* operand bit 17 through bit 65 (bit 64 and bit 65 are forced to 0's) and S*j* and V*j* bit 17 through bit 65 (bit 64 and bit 65 are forced to a 0's).

The NG option also detects exceptional inputs such as:

- Zero *j* exponent/fraction
- Zero *k* exponent/fraction
- Signaling NAN *j* operand
- Signaling NAN *k* operand
- Quiet NAN *j* operand
- Quiet NAN *k* operand
- Infinite NAN *j* operand
- Infinite NAN *k* operand

and communicates the presence of these inputs to the NF options and the NH option.

NH Option

The NH option performs the final summation for the floating-point multiply pyramid and sends the final coefficient and exponent to the result registers. The NH also transmits the interrupt signal to the AY option where it is relayed to the HH option for use in the exchange package.

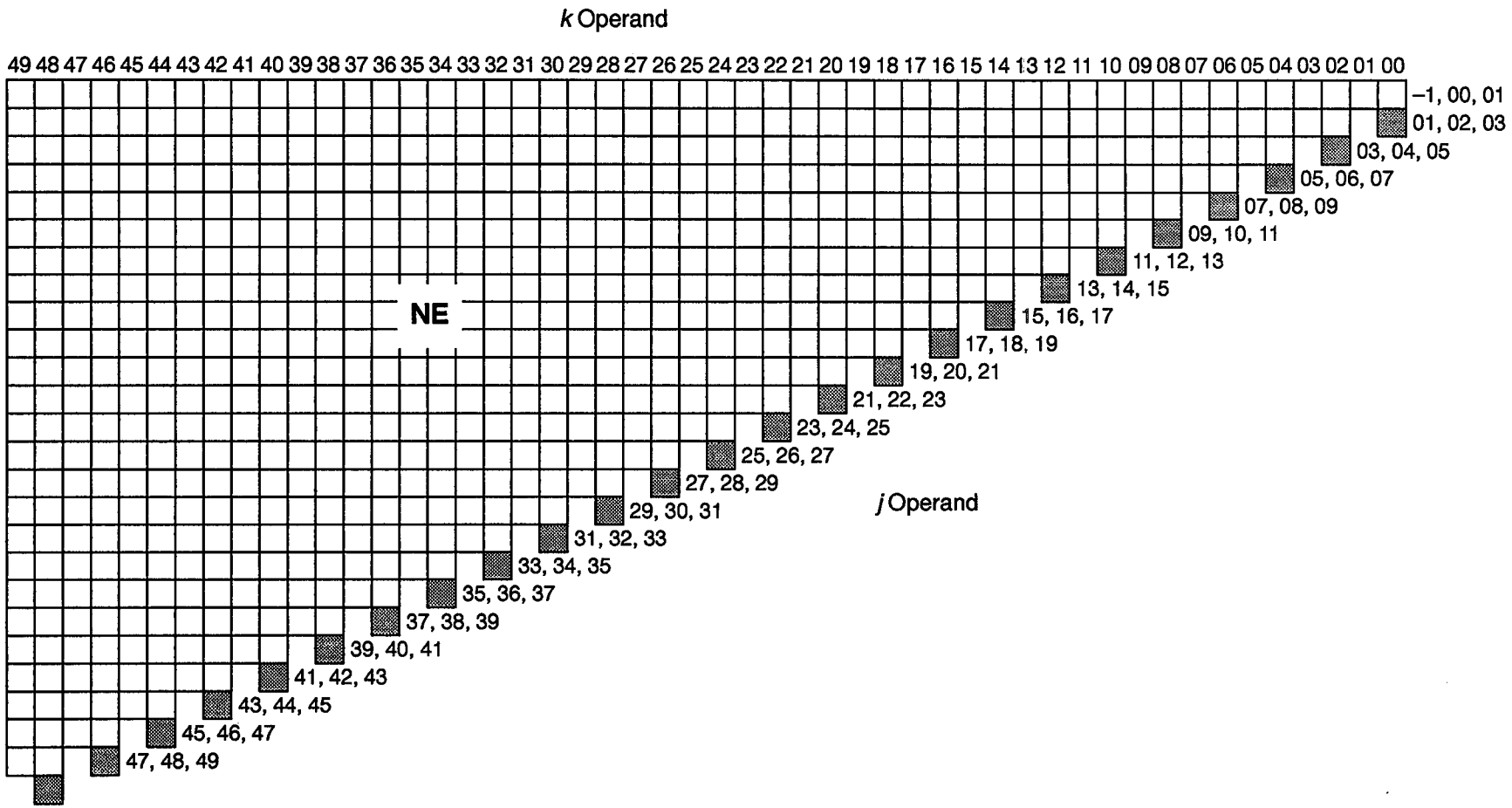


Figure 69. NE Option Pyramid

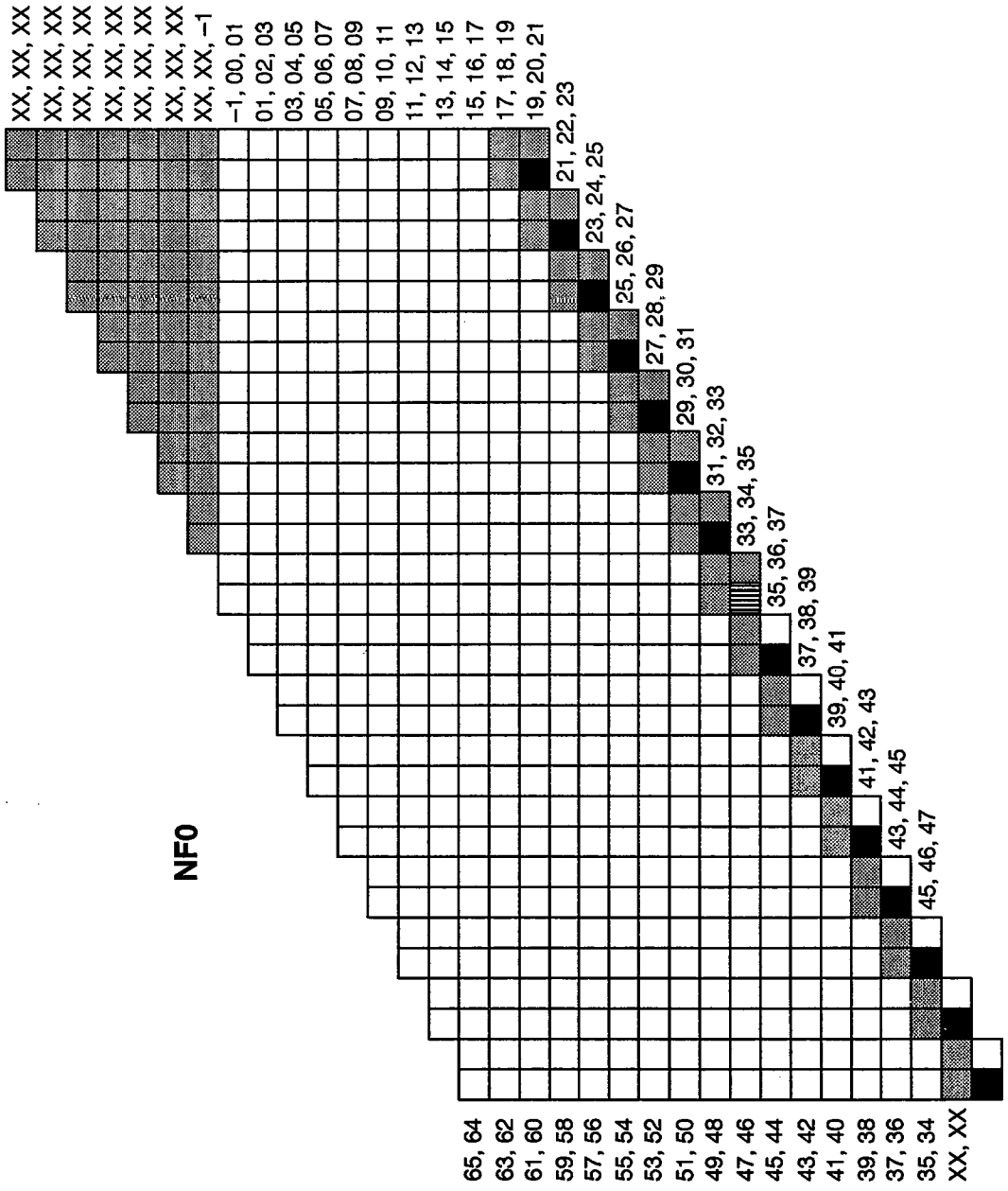


Figure 70. NF0 Option Pyramid

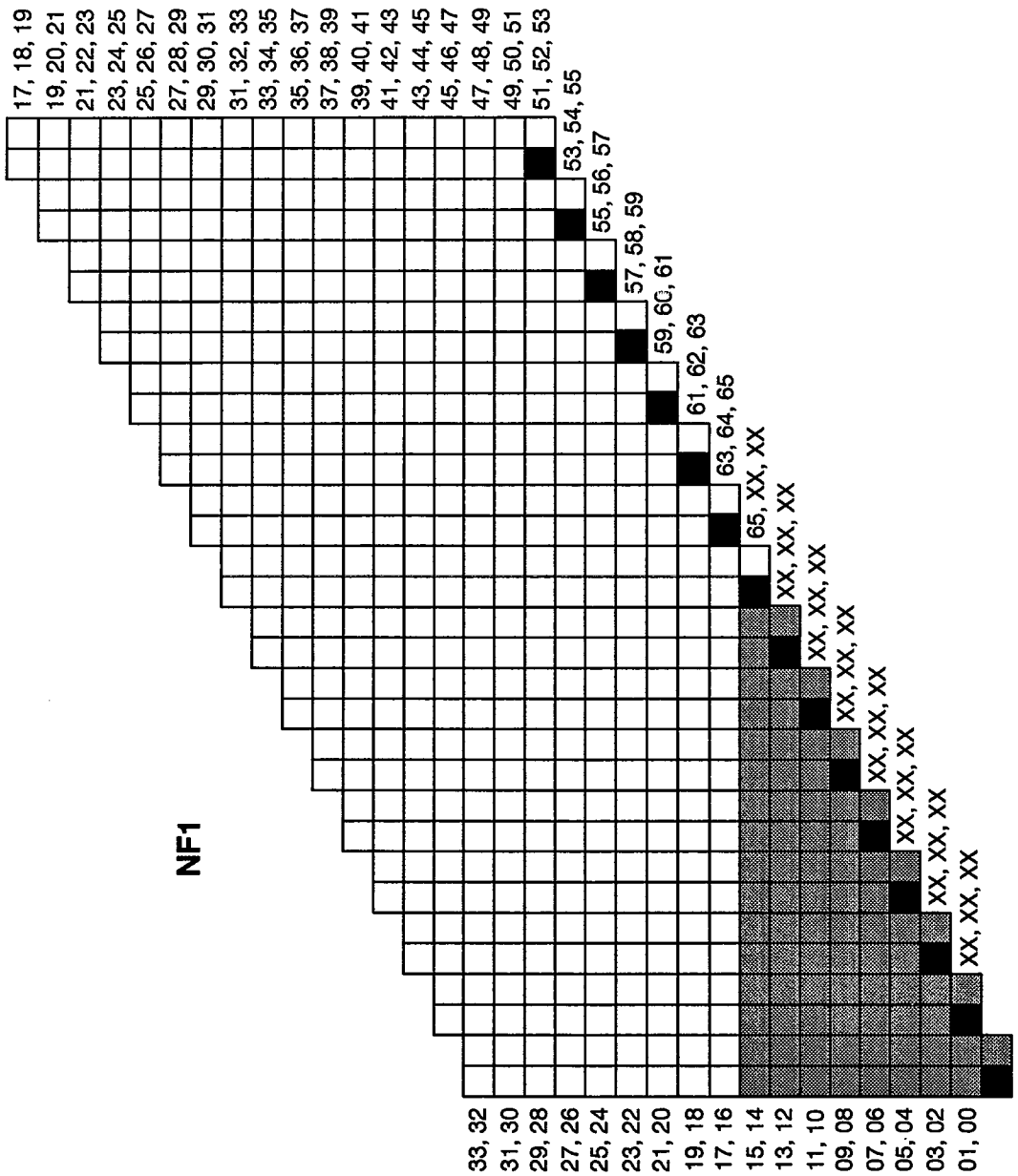


Figure 71. NF1 Option Pyramid

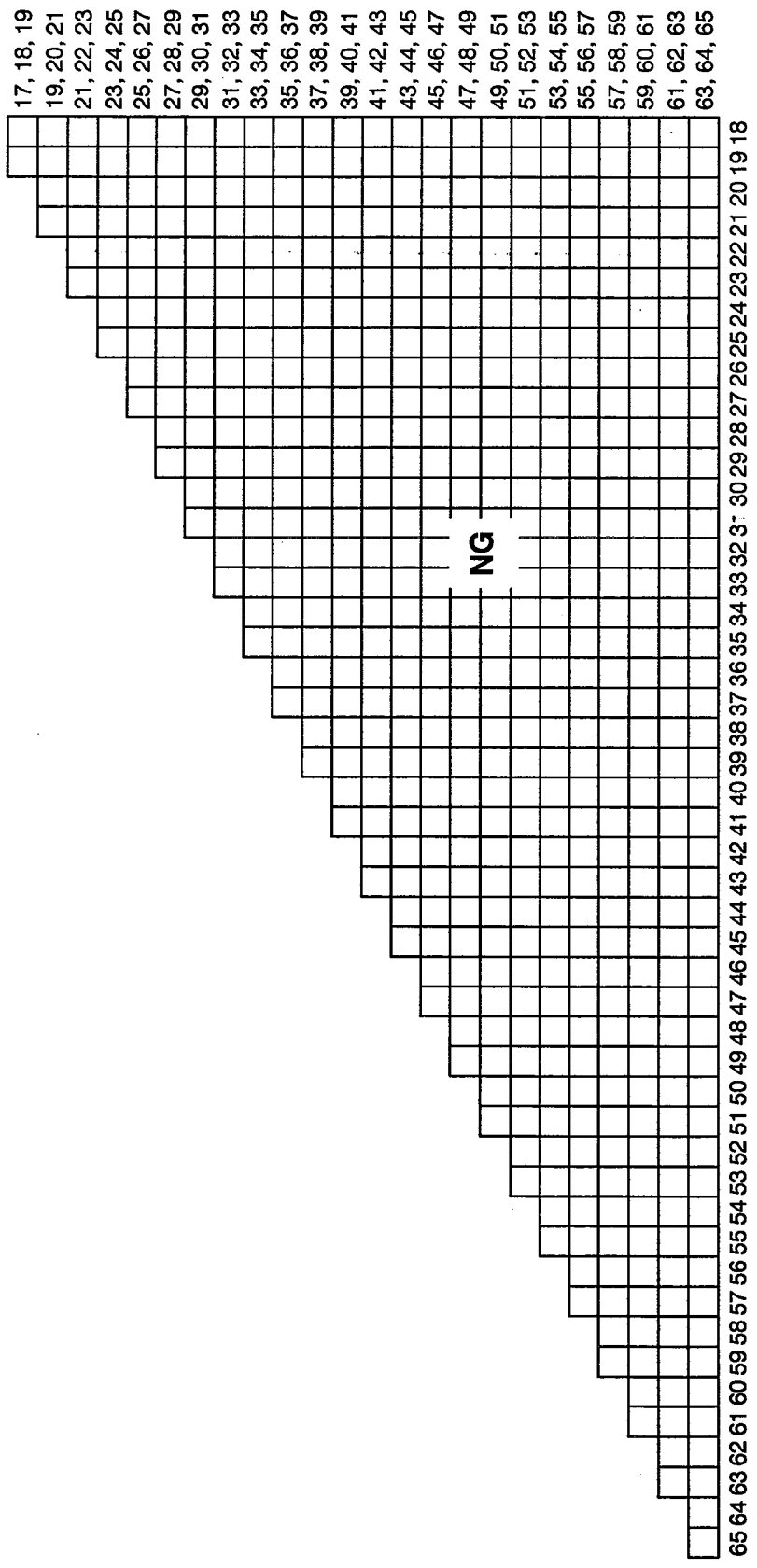


Figure 72. NG Option Pyramid

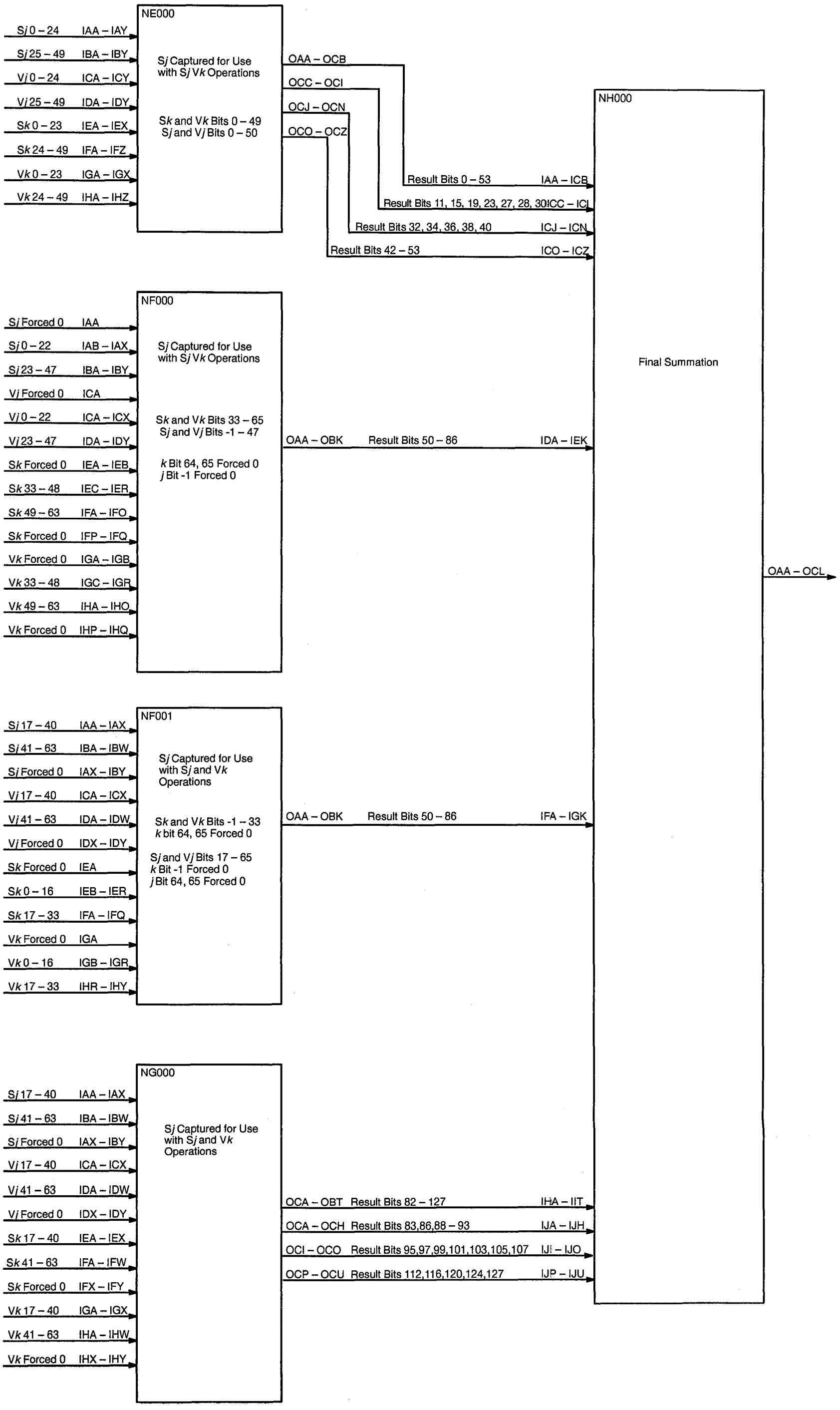


Figure 73. Multiply Data Paths

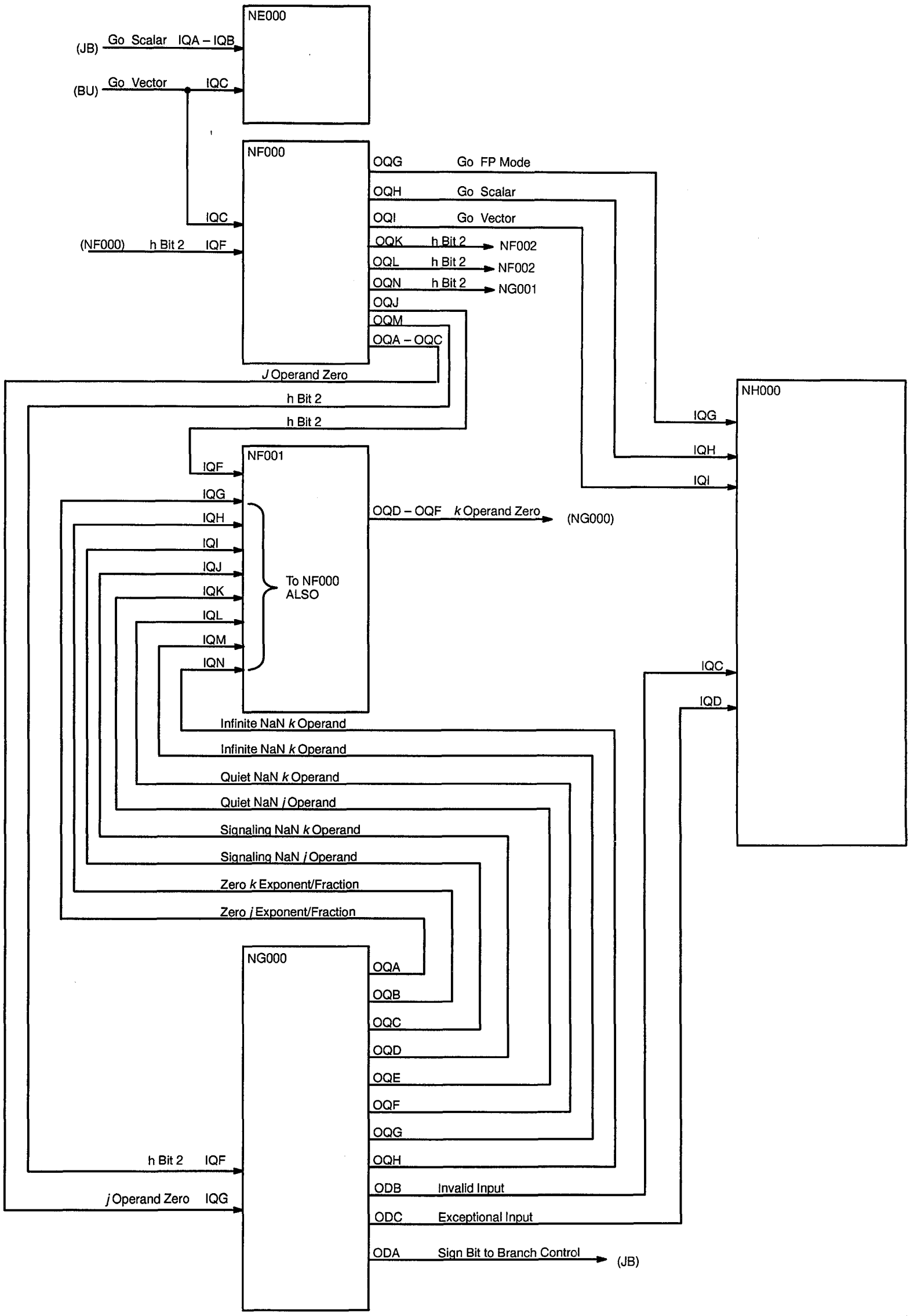


Figure 74. Multiply Control Paths

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 loads the B array with the V*j* operand. The second function performs the A x 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 x B^T operation, each OA produces a partial result for each of the 32 elements. The partial results are then sent to 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, which results 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 75 is an illustration of 20 x 20 and 50 x 50 matrices stored in vector registers.

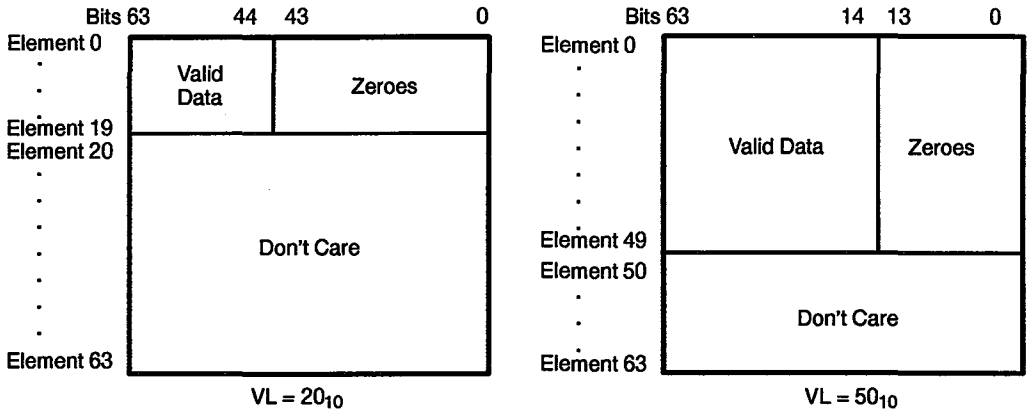


Figure 75. Vector Storage of Bit Matrices

In this section, the notation used to represent individual bits of a matrix is 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

Matrices A and B can be represented mathematically as illustrated in Figure 76. Note that the ultimate degree of both element and bit can be represented by *n* because matrices must be square. Each row of a matrix corresponds to an element of a vector register.

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

Figure 76. 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 illustrated in Figure 77.

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

Figure 77. B Matrix and B^t Matrix Relationships

The operation, $C = AB^t$, is illustrated in Figure 78.

$$AB^t = \begin{vmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ . & . & . & & . \\ . & . & . & & . \\ . & . & . & . & . \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \\ & A & & & \end{vmatrix} \begin{vmatrix} b_{11} & b_{21} & b_{31} & \cdots & b_{n1} \\ b_{12} & b_{22} & b_{32} & \cdots & b_{n2} \\ b_{13} & b_{23} & b_{33} & \cdots & b_{n3} \\ . & . & . & & . \\ . & . & . & & . \\ . & . & . & & . \\ b_{1n} & b_{2n} & b_{3n} & \cdots & b_{nn} \\ & B^t & & & \end{vmatrix} = \begin{vmatrix} c_{11} & c_{12} & c_{13} & \cdots & c_{1n} \\ c_{21} & c_{22} & c_{23} & \cdots & c_{2n} \\ c_{31} & c_{32} & c_{32} & \cdots & c_{3n} \\ . & . & . & & . \\ . & . & . & & . \\ . & . & . & & . \\ c_{n1} & c_{n2} & c_{n2} & \cdots & c_{nn} \\ & C & & & \end{vmatrix}$$

where:

$$C_{11}=a_{11}b_{11}\oplus a_{12}b_{12}\oplus a_{13}b_{13}\oplus \ldots \oplus a_{1n}b_{1n}\ \dagger$$
$$C_{12}=a_{11}b_{21}\oplus a_{12}b_{22}\oplus a_{13}b_{23}\oplus \ldots \oplus a_{1n}b_{2n}$$
$$C_{13}=a_{11}b_{31}\oplus a_{12}b_{32}\oplus a_{13}b_{33}\oplus \ldots \oplus a_{1n}b_{3n}$$
$$\vdots$$
$$C_{21}=a_{21}b_{11}\oplus a_{22}b_{12}\oplus a_{23}b_{13}\oplus \ldots \oplus a_{2n}b_{1n}$$
$$\vdots$$
$$C_{32}=a_{31}b_{21}\oplus a_{32}b_{22}\oplus a_{33}b_{23}\oplus \ldots \oplus a_{3n}b_{2n}$$
$$\vdots$$

† $\oplus$ indicates an exclusive OR operation.

Figure 78. Multiplication of A and B^t

Instructions

Refer to Table 45 for a list of the bit matrix multiply instructions.

Table 45. Bit Matrix Multiply Instructions

Instruction	CAL	Description
1740*j*4	BMM LV*j*	Transmit V*j* elements 0 – 63 to B matrix
1740*j*5 †	BMM UV*j*	Transmit V*j* elements 64 – 127 to B matrix
174*ij*6	V*i* V*j* * BT	Transmit the value of V*j* multiplied by the transposed B matrix to V*i*
070*ij*6	S*i* S*j* * BT	Transmit the value of S*j* multiplied by the transposed B matrix to S*i*
002210	CBL	Clear the bit matrix loaded (BML) flag

† New instruction

Refer to Figure 79 for a BMM block diagram for pipe 0 and to Figure 80 for a BMM block diagram for pipe 1.

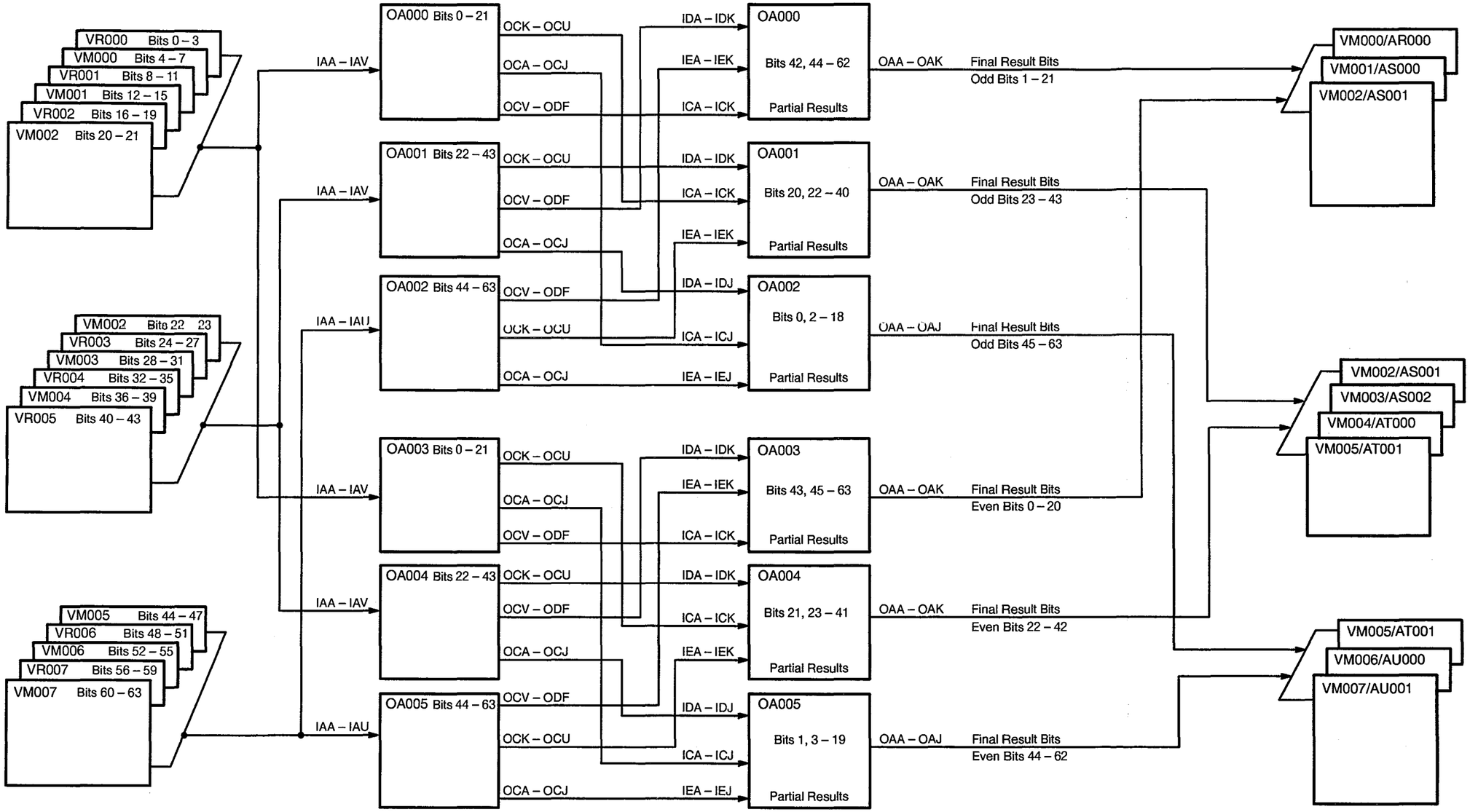


Figure 79. Bit Matrix Multiply Block Diagram, Pipe 0

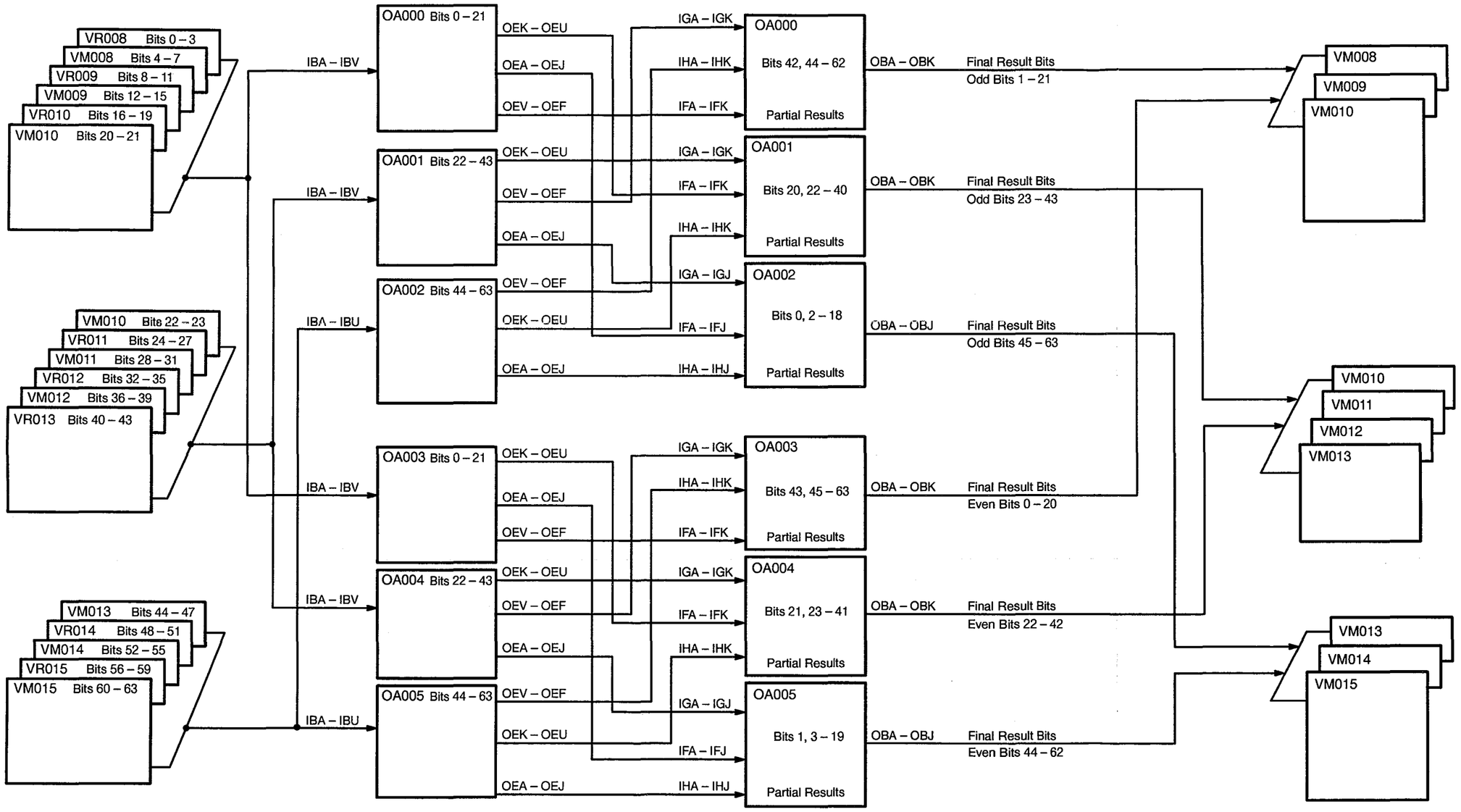


Figure 80. Bit Matrix Multiply Block Diagram, Pipe 1

INSTRUCTION BUFFERS

The instruction buffers are distributed across four IC options. (Table 46 illustrates 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 the functions listed in Table 46.

Table 46. 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

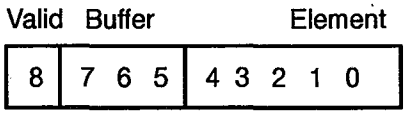
Fetch

The IC options generate a deadstart fetch after the first 20_8 words (the number of words in the exchange package) have been received. The IC option counts the number of common memory valid codes received, and this count enables the generation of the deadstart fetch signal.

When data is fetched from memory, it is requested as a block of 32 words (4 blocks of 8 words where the first word of this block is 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 addresses 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, and because of the memory code, it is reordered inside the buffer.

A 9-bit code accompanies every 16 bits of memory data. 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.



Two words of data arrive together at the IC options. As the data starts to arrive, the IC options sense the first 4 words. These words proceed through a bypass path, to the read-out registers, and then to the JB 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, issue stops until the buffers fill and the buffer valid bit sets. The instruction parcels are then transmitted to the JB options from the buffers.

Prefetch

A prefetch begins when the buffer read-out pointer reaches address 30_8 in the buffer or a branch occurs to addresses 30 to 37_8.

The prefetch determines if the next sequential buffer is already in-stack. If it is not, a fetch accesses the next sequential common memory address. When the count in the buffer reaches 37_8, the IC advances the buffer pointer and ensures 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 at times degrade 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 awaiting prefetch, 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 program 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 and the branch is within 8 words of the last valid LAT address, a range error may occur.

Refer to Figure 81 for the IC options bit layout, to Figure 82 for an IC block diagram, and to Figure 83 for the IC option terms.

Figure 84 is a block diagram of the memory-to-instruction buffers for path 1, and Figure 85 is a block diagram of the memory-to-instruction buffers for path 2. Figure 86 is a block diagram of the common memory path code 1 fanouts, and Figure 87 is a block diagram of the common memory path code 2 fanouts.

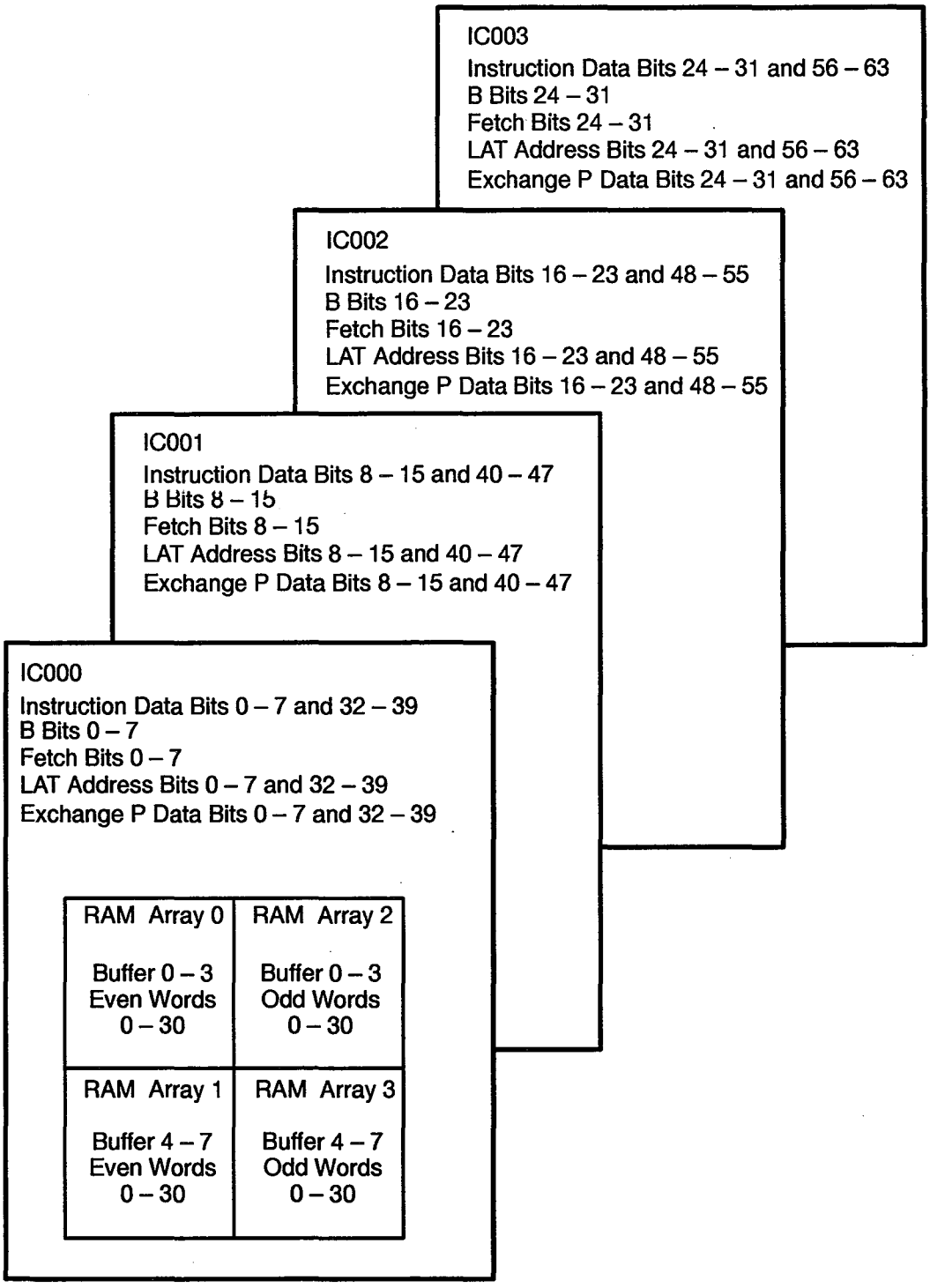


Figure 81. IC Options Bit Layout

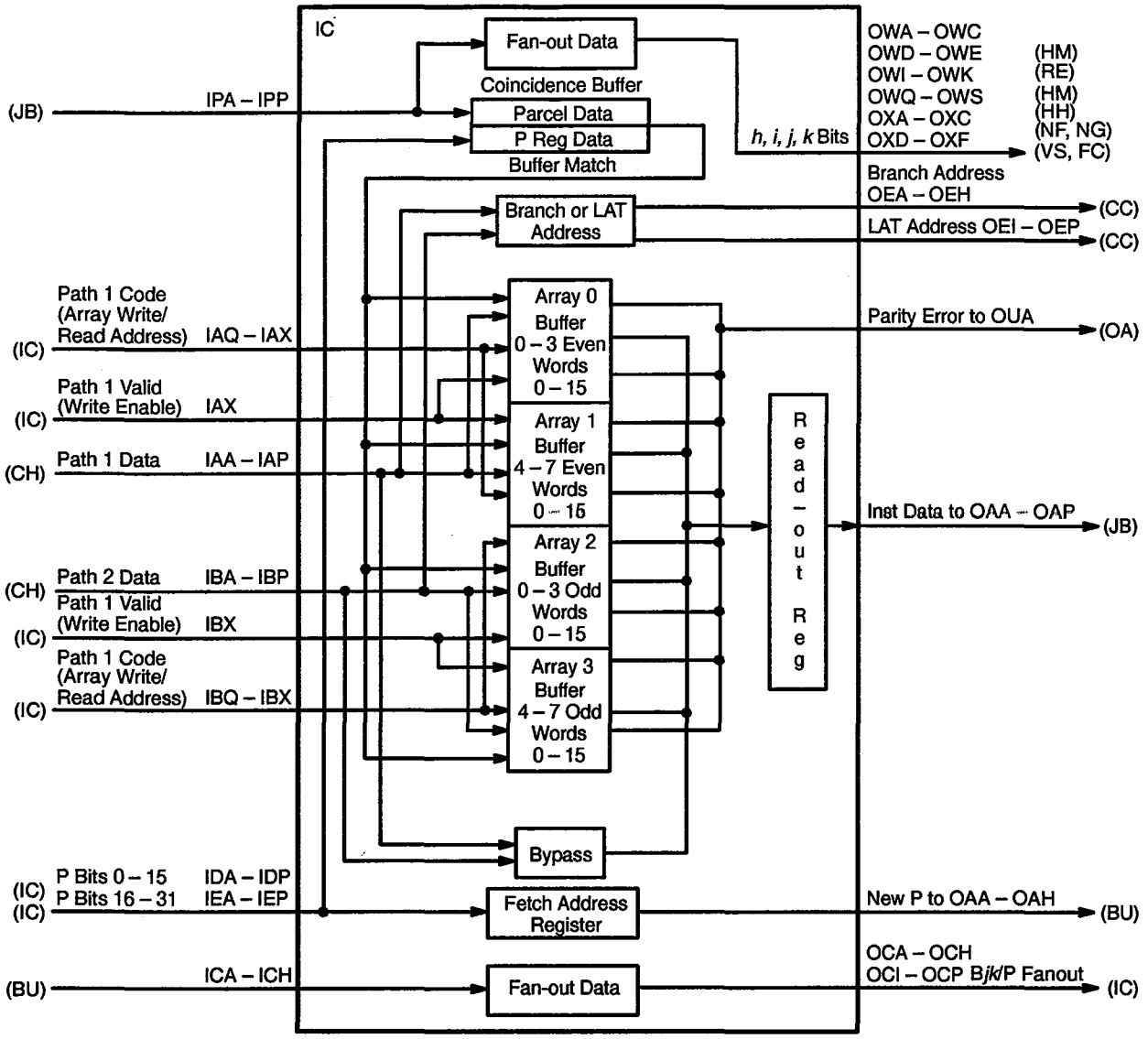


Figure 82. IC Block Diagram

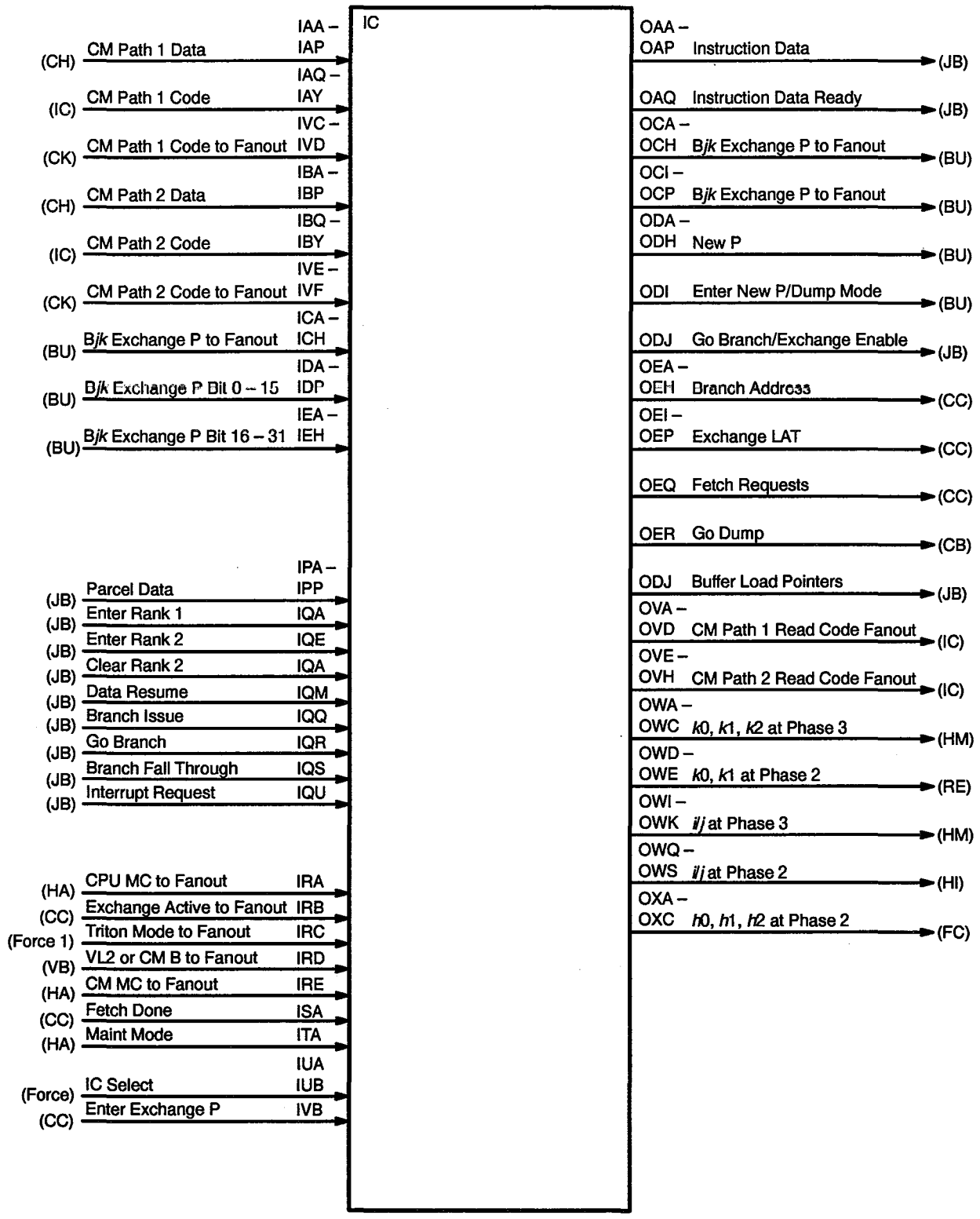


Figure 83. IC Option Terms

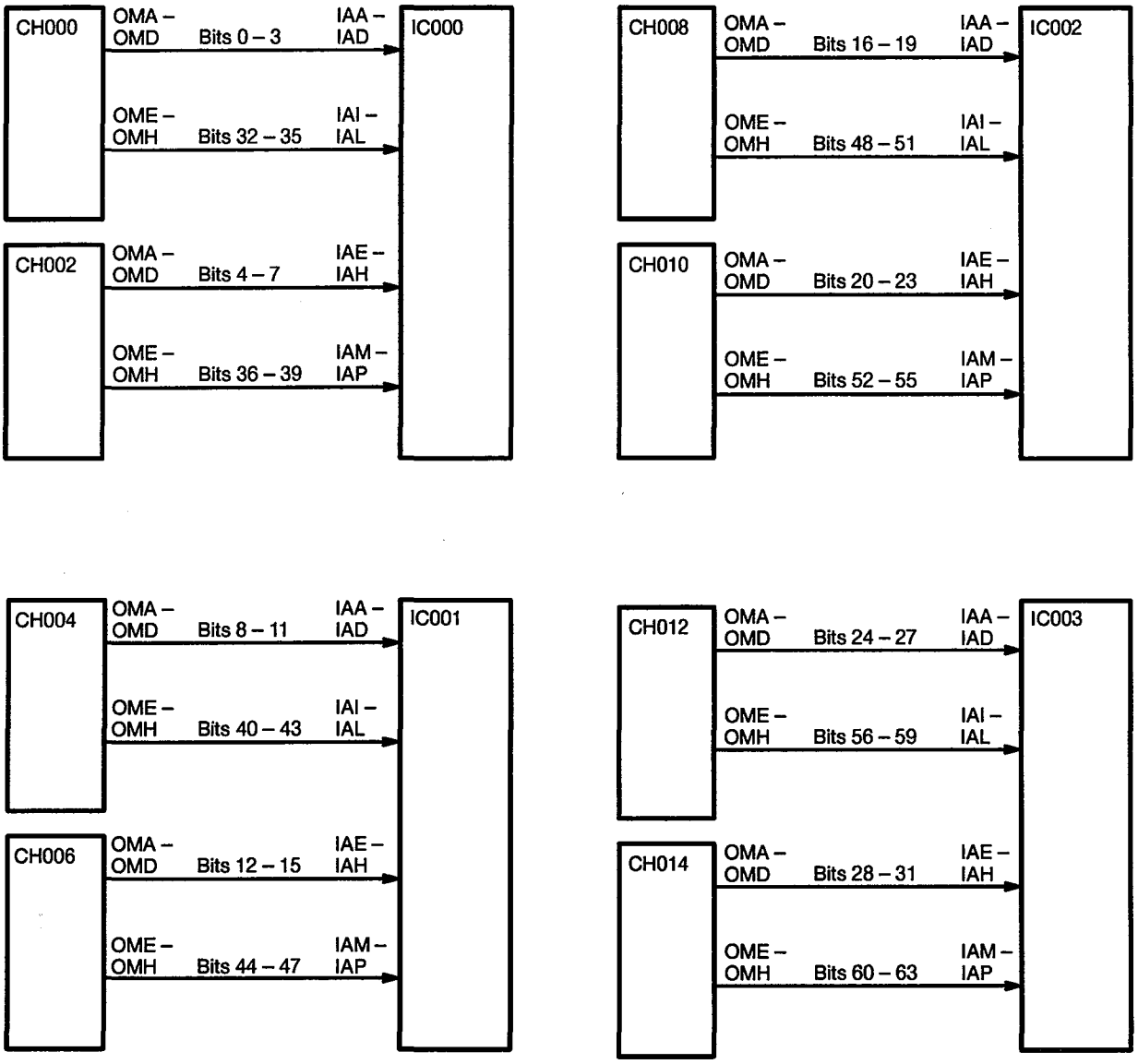


Figure 84. Memory-to-instruction Buffers, Path 1

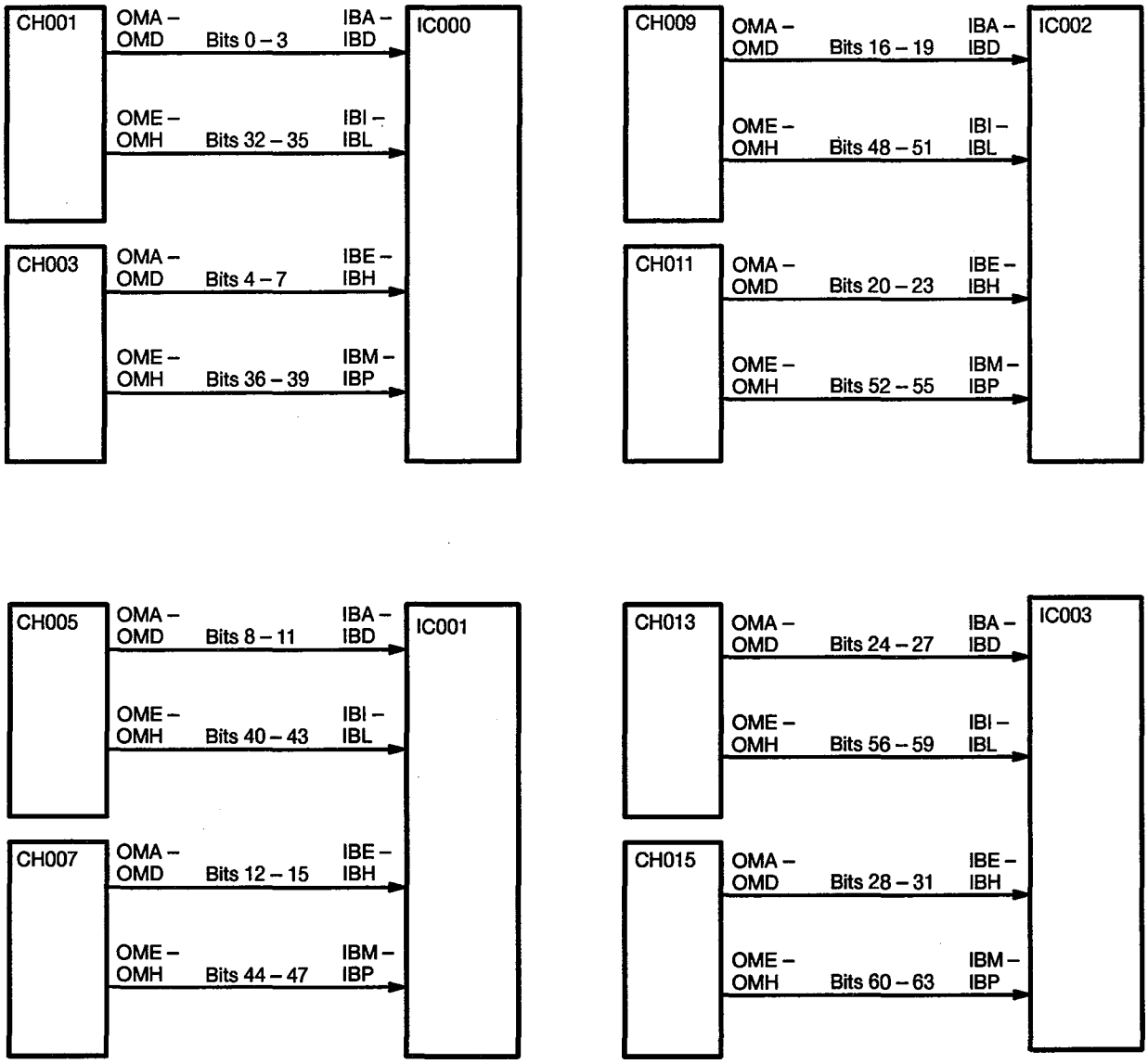


Figure 85. Memory-to-instruction Buffers, Path 2

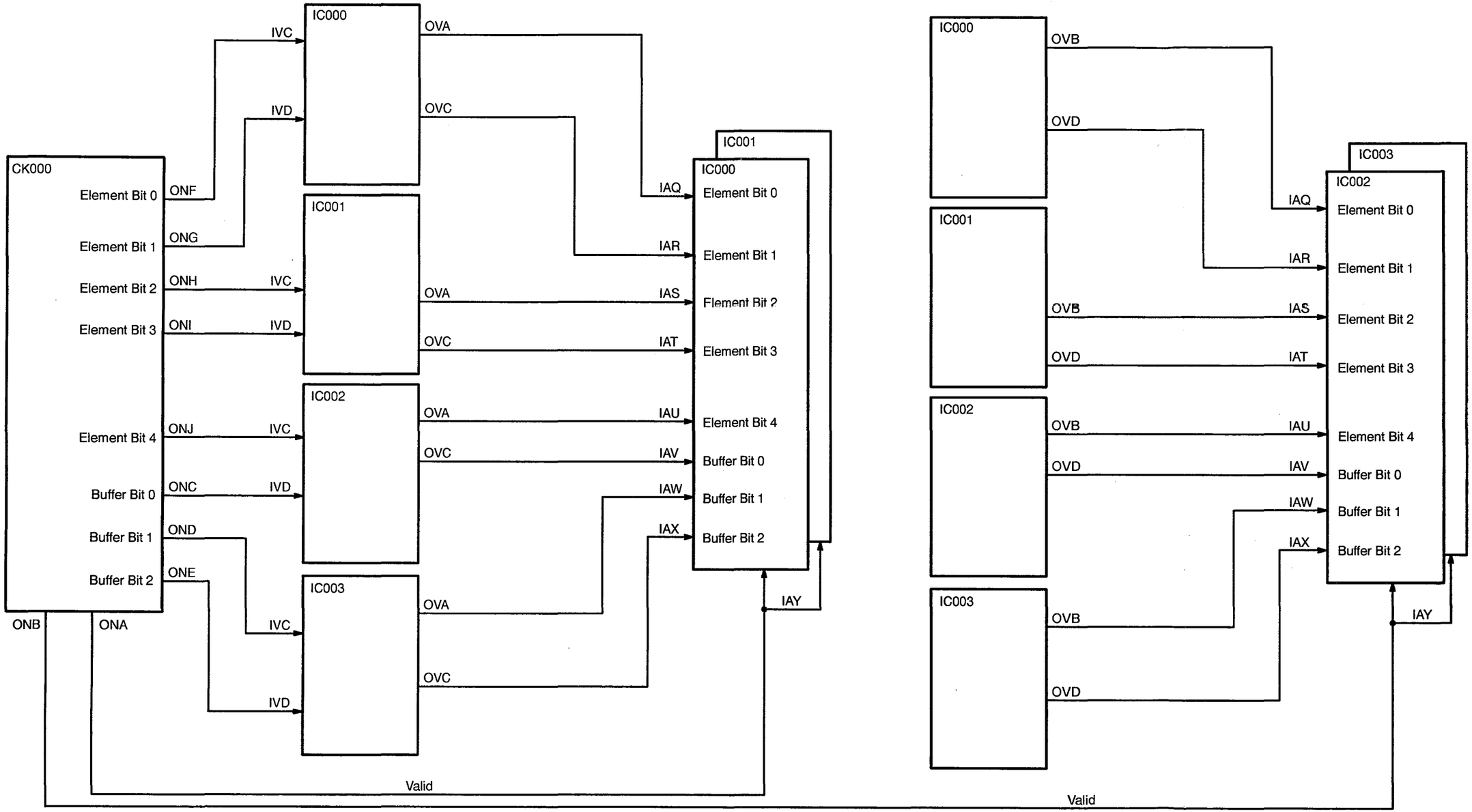


Figure 86. Common Memory Path, Code 1 Fanouts

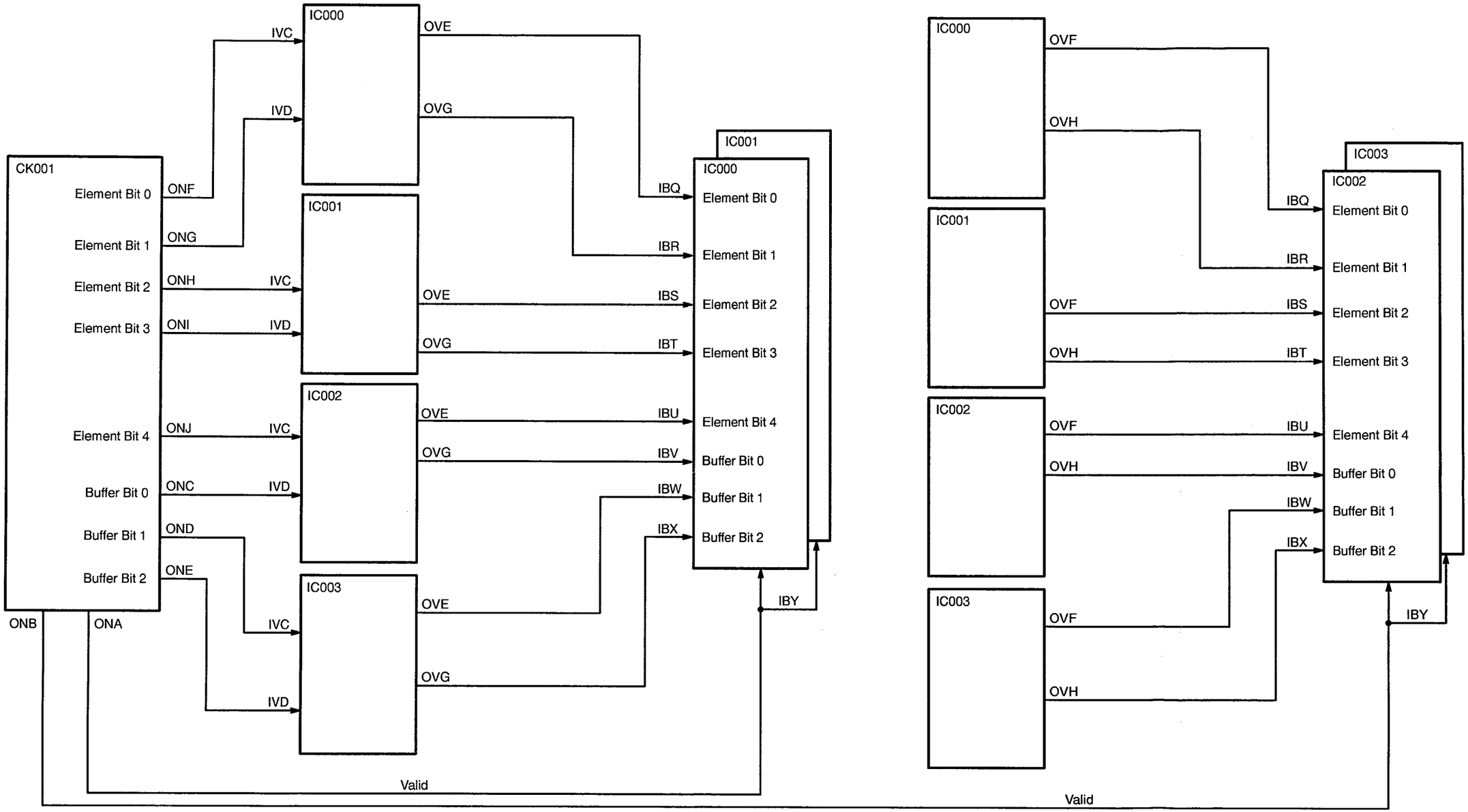


Figure 87. Common Memory Path, Code 2 Fanouts

INSTRUCTION ISSUE

In the CRAY T90 series computer system, a process called instruction issue introduces instructions into the central processing unit (CPU).

The first instruction parcel is read from of 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 88 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 on to CIP as the other three parcels are leaving. In the next clock period, the fourth parcel leaves CIP, and the value in the P register increments by 4.

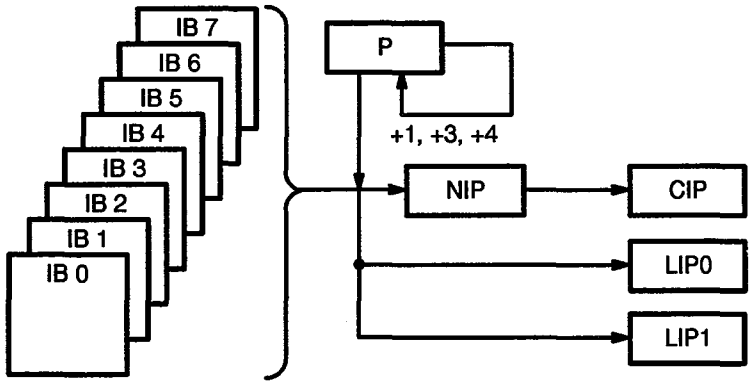


Figure 88. Instruction Issue Block Diagram

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 examined 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 or 0055*jk*).

Figure 89 illustrates the format of a 1-parcel instruction.

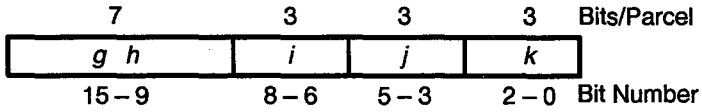


Figure 89. Format for a 1-parcel Instruction

Three-parcel Instructions

In the 3-parcel instruction format, the *nm* fields hold the 32-bit address or constant value. Figure 90 illustrates a 3-parcel instruction format.

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

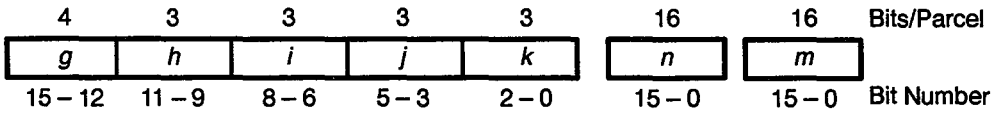


Figure 90. Format for a 3-parcel Instruction

Four-parcel Instructions

In the 3-parcel instruction format, the instruction field mnemonic *pmn* represents a 48-bit field of which the *p* portion is the most significant parcel. Figure 91 illustrates a 4-parcel instruction format.

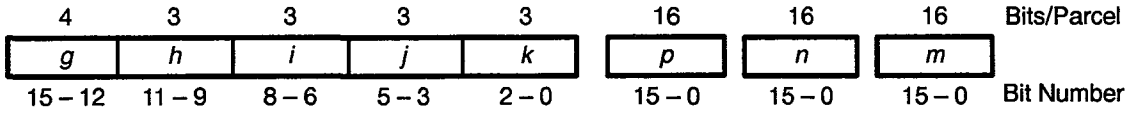


Figure 91. Format for a 4-parcel Instruction

Four-parcel instructions are used in A and S register memory references that use extended addressing. The *h* field selects an A register that contains an address index. The *i* field designates which A or S register is the source or destination of the data. During read references, bit 1 of the *j* field 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.

Instruction Decode

When an instruction parcel is loaded into NIP, its size is determined. 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 BU0 and BU1 options. The P register indicates 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 B*jk* as a result of a 005*ijk* instruction
- Load from the *ijk* or *nm* fields of a 006*ijk*, 007*ijk*, or 01*xjk* instruction

Every time a parcel issues, the JB option sends an Advance P signal to the BU options. Advance P increments the P register by 1.

Coincidence

A condition called *coincidence* exists when the next needed parcel is in one of the eight instruction buffers. (Coincidence is checked only on branch instructions.) A coincidence check compares the upper 25 bits of the P register to the 25-bit buffer address (A) register and 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.

Reading the Instruction Buffer

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

JB Option

The two JB options on the CP module 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 JB option also has all the resource reservations and holds issue if a resource is busy. The JB options are responsible for the functions described in the following subsections.

Parcel Data Distribution

The JB option transmits parcel data to the AV, AW, AX, AY, BU, and VB options and alters the *j* field going to the AV, AW, AX, and AY options for certain instruction types during the following instructions:

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

The JB 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 47 for a list of these codes.

Table 47. Read-out Path Codes

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

Table 47. Read-out Path Codes (continued)

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

A/S/V/B/T Register Requests

The JB 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 JB option also receives a vector read/write (R/W) release for V registers and a B/T read/write release. The JB option also transmits A and S register entry codes. The A and S registers use these codes, the *ghijk* field, the instruction, and the 2-bit register read-out code to define the instruction to be performed and to reserve the needed path.

Functional Unit Requests

The JB option detects 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 / 070*ij*1 / EIS 153*ij*0,1
- Vector Shift: 150 – 153
- Vector Add: 154 – 157
- Floating Multiply/Divide: 160 – 167
- Floating Add: 170 – 173

- Square Root: 070*ij*1, 174*ij*0 (V pop, parity, leading zero, iota: 174*ij*(1 – 3)
- Matrix Multiply: 174*ij*(4 – 7) / 070*ij*(6 – 7)

Constant Data Requests

The JB option checks for the presence of constant data in multiple-parcel instructions such as jumps, branches, and instructions that use the *pmn* fields. Each JB option handles 32 bits of the constant data distribution. JB0 transmits data to the AV, AW, and CD options through the A series options; and JB1 transmits data to the AX, AY, and CD options through the A series options. JB0 also provides the *jk* data on the constant path when needed.

Extended Instruction Set (EIS) Requests

When the JB option issues 005400 or 0055*xx* instructions, the parcel following either of these instructions is defined by the extended instruction set. If an EIS-capable instruction is issued without a preceeding 005400 or 0055*xx* instruction, the instruction issues and performs its primary function. For example:

044*ijk* Transmit the logical product of (S*j*) and (S*k*) to S*i*

044*ijk* In EIS mode, this instruction transmits the logical product of (A*j*) and (A*k*) to A*i*

Common Memory Requests

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

- Release Port A
- Release Port B
- Release Port C
- Bidirectional Mode: (Mode = 1) Enables block reads and writes at the same time

- Common Memory Quiet: Indicates that all memory activity in the CPU has been completed. 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 JB 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 JB options check the conditional branch test conditions to determine whether the condition is satisfied; if it is, the JB option issues a Go Branch signal to the IC options.

Exchange Requests

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

- 000000 (error exit) issues. Issue stops, P advances.
- 0040*jk* (exit *k*) issues. Issue stops, P stops.

- The shared path is released. The state of Go Semaphore Branch determines whether P advances on a 0040*jk*. One of two possible results can occur:
  - A normal exit occurs and P advances when the shared path is released and Go Semaphore Branch is a 0.
  - 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 HH, 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 JB option performs the phase 1 checks and stops issue. If the phase 2 checks are all valid, the JB 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) has 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*) has issued, a release path must occur to clear the JB option control.

Issue will resume when Go Branch occurs.

Control Signal Distribution

The JB option transmits the following control signals:

- Issue group 0, 1, and 2: These signals are combined on the BU and VA options to complete the issue signal.
- Issue: Sent to the AN option for fanout.
- Enter Vector Length: Sent to the AV option following the decode of a 00200*k* (A*k* to VL) instruction.
- Read Vector Mask: Sent to the SS option during a 073*i* (0 – 3) 0 (VM0 or VM1 to S*i* or A*i*) instruction.
- Enter Vector Mask: Sent to the SS option during a 0030*j* (0 – 3) (S*i* or A*i* to VM0 or VM1) instruction.
- Go Scalar Pop/Parity/Lz: Sent to the SS option during a 026*ij* (0 – 3) or 027*ij* (0 – 1).
- Go Scalar Double Shift: Sent to the SS option during a 056*ijk* Shift (S*i*) and (S*j*) left A*k* places to S*i*.
- Go A Type: Sent to the SS option when a 005400 (EIS) is issued using A register data.
- Go Scalar Divide: Sent to the RE option during a 065*ijk* instruction.
- Go Scalar Floating Add: JB1 sends this signal to the FC option when a 062*ijk* (sum) or 063*ijk* (difference) issues.
- Go Scalar Floating Multiply: Sent to the NG option when a 064*ijk* instruction issues.
- Go Address Multiply: Sent to the AV option when a 032*ijk* issues.
- Go Compare: This signal is transmitted to the FC option from JB001 when a 00550x 164*ijk* issues.
- Common Memory A or S Requests: Sent to the CD options when a memory load or store issues. JB0 sends out an A register request, and JB1 sends out S register requests.
- Common Memory A or S Writes: Sent to the CD options when a memory write 11*hi*xx*pnm* or 13*hi*xx*pnm* issues. JB0 sends out A register write requests, and JB1 sends out S register write requests.

- CM Port B Enabled: Sent to the VB option through the JB0 option and to the BU option through the JB1 options to select the vector read ports.
- Vector Logical #2 Enabled: Sent to the VB options by JB0 to select vector logical functional units.
- Data Resume: Sent to the instruction stack (IC options) to indicate that the JB option can accept another word.
- Go Exchange: Sent to the IC options to indicate that an exchange is required. Another copy is sent to the HH option to clear the SIE bit (taking I/O interrupt), and to the CC option to begin the swapping of exchange packages in memory.
- Go Branch: Sent to the IC options to indicate that a conditional branch condition has been satisfied.
- Branch Fall Through: Sent to the IC options to indicate that a conditional branch has failed the condition test.
- Branch Issued: Sent to the IC options to indicate that a branch has issued.
- Enter Rank 1, Enter Rank 2, or Clear Rank 2: 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 (HI) 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: Sent to the P register (BU options) to advance P by 1 as each parcel is issued.

Branch Instruction Control

The JB 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 satisfying a condition 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. The unconditional jump instruction (006100*mn*) branches to the address in *nm* field.

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.

The 007100*nm* jump instruction is an indirect jump. This 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 HI000, HH000, and HH001. The CIP located on the HI options is responsible for the instructions that affect the exchange package and performance monitor.

The HH option CIP is used for A/S path release and provides A/S*i* designators and shared path release. The JB options determine whether any register or functional unit reservation exists. If not, these options send the **Issue** signal to the HH and HI options. The instruction issues, reserving the appropriate registers and/or functional unit. If resource conflicts do exist, the JB 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 JB 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 JB options to indicate that the units are available.

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

Refer to Figure 92 through Figure 98 for related instruction issue block diagrams.

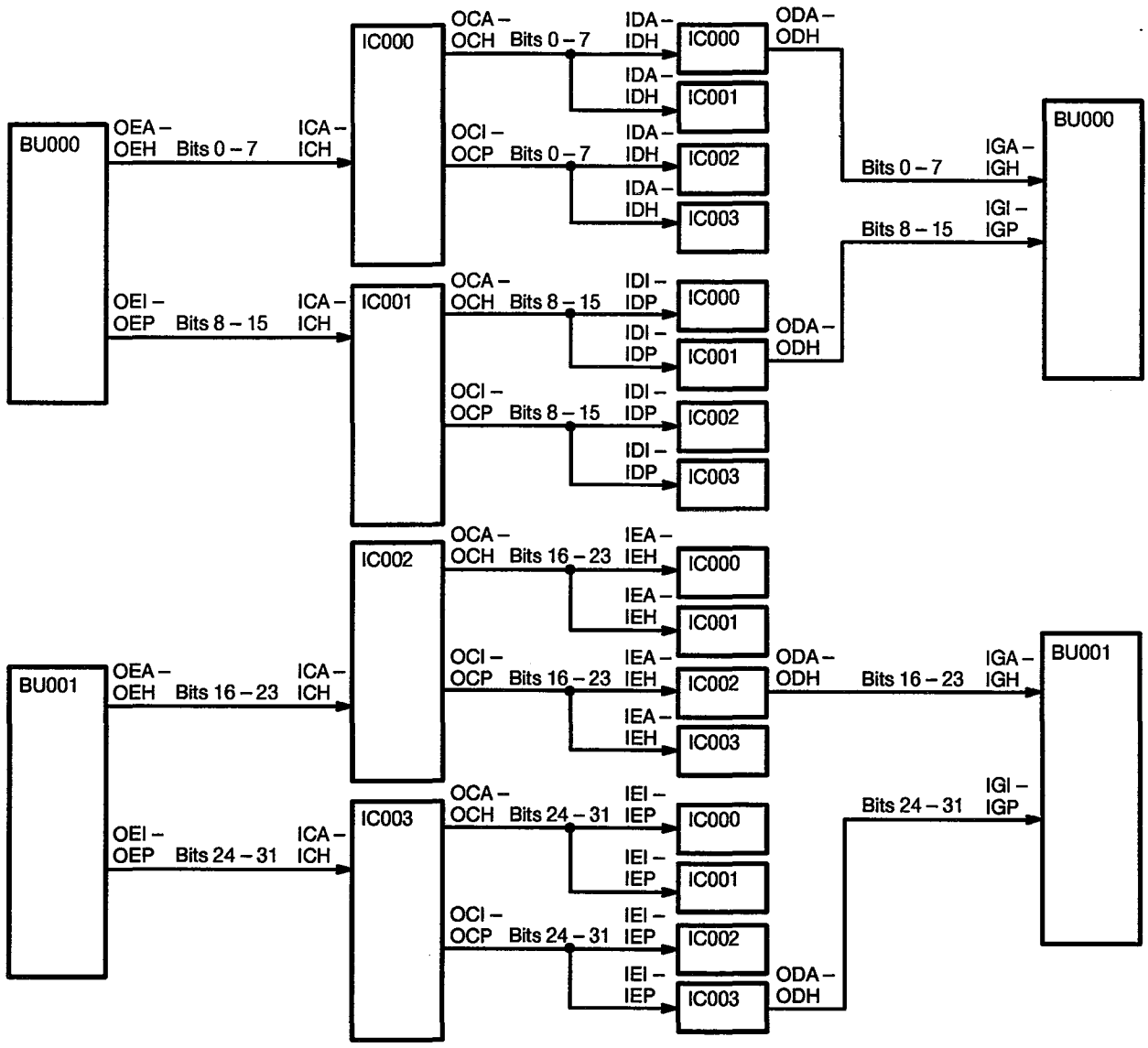


Figure 92. B*jk* (Exchange P) Fan-out Bits

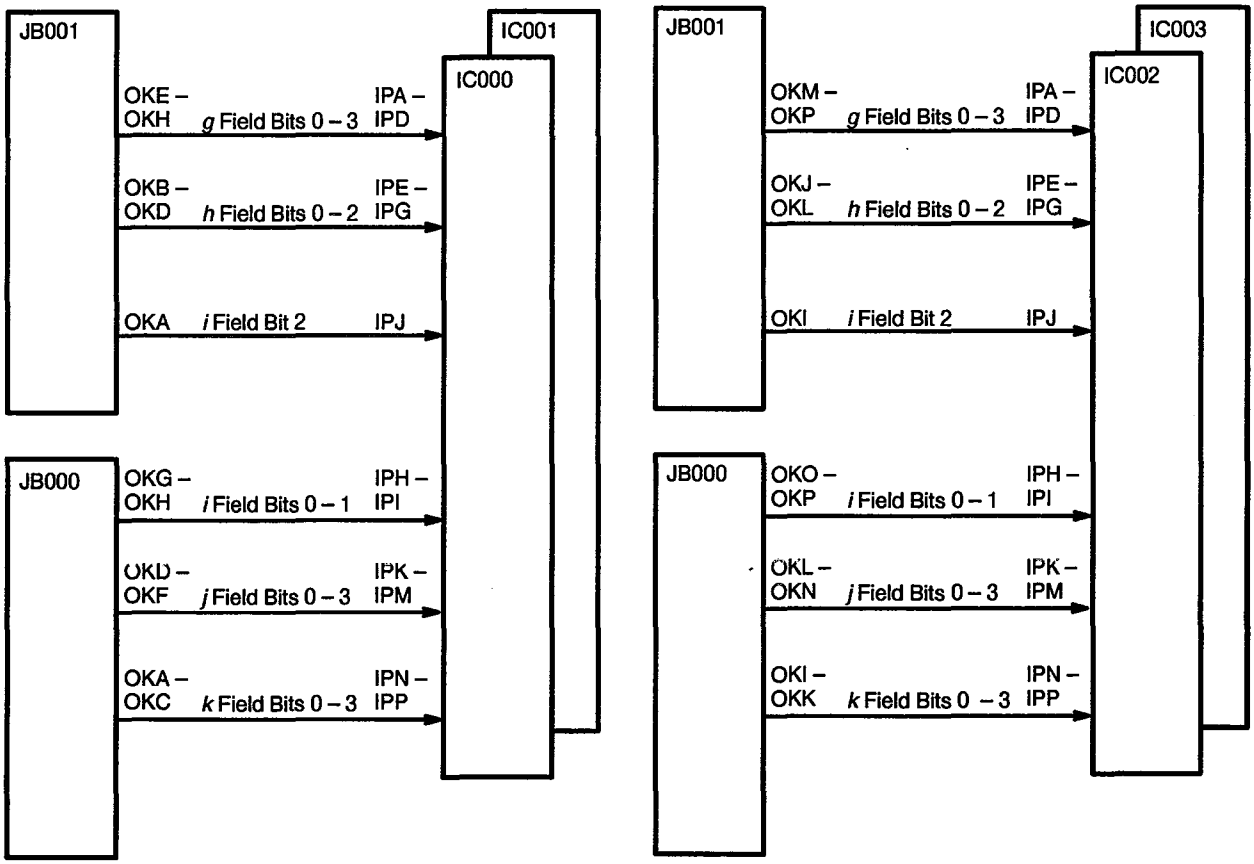


Figure 93. JB-to-IC Parcel Data for Branches

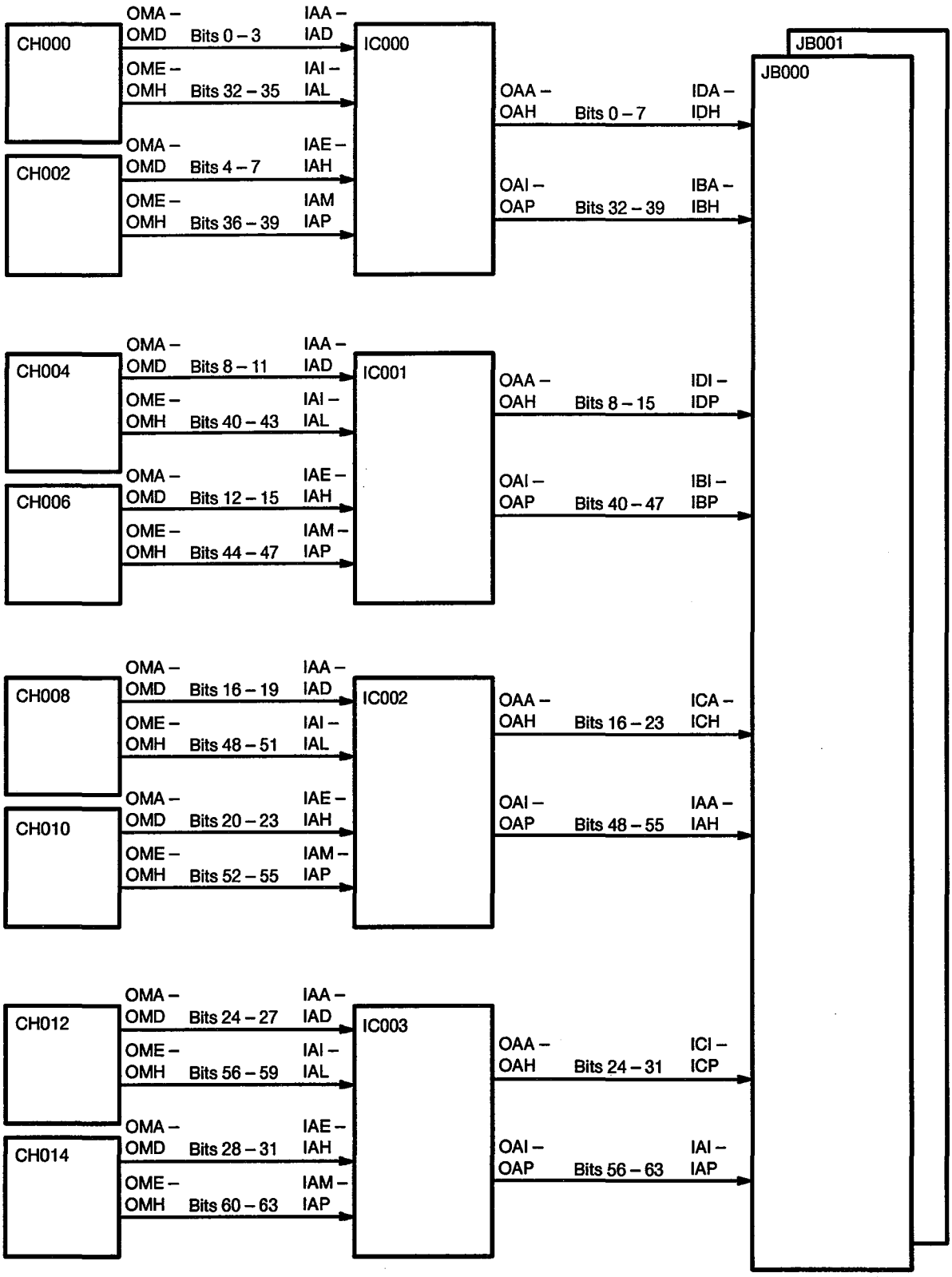


Figure 94. Path 1 CH-to-IC-to-JB Option

Figure 95. Path 2 CH-to-IC-to-JB Option

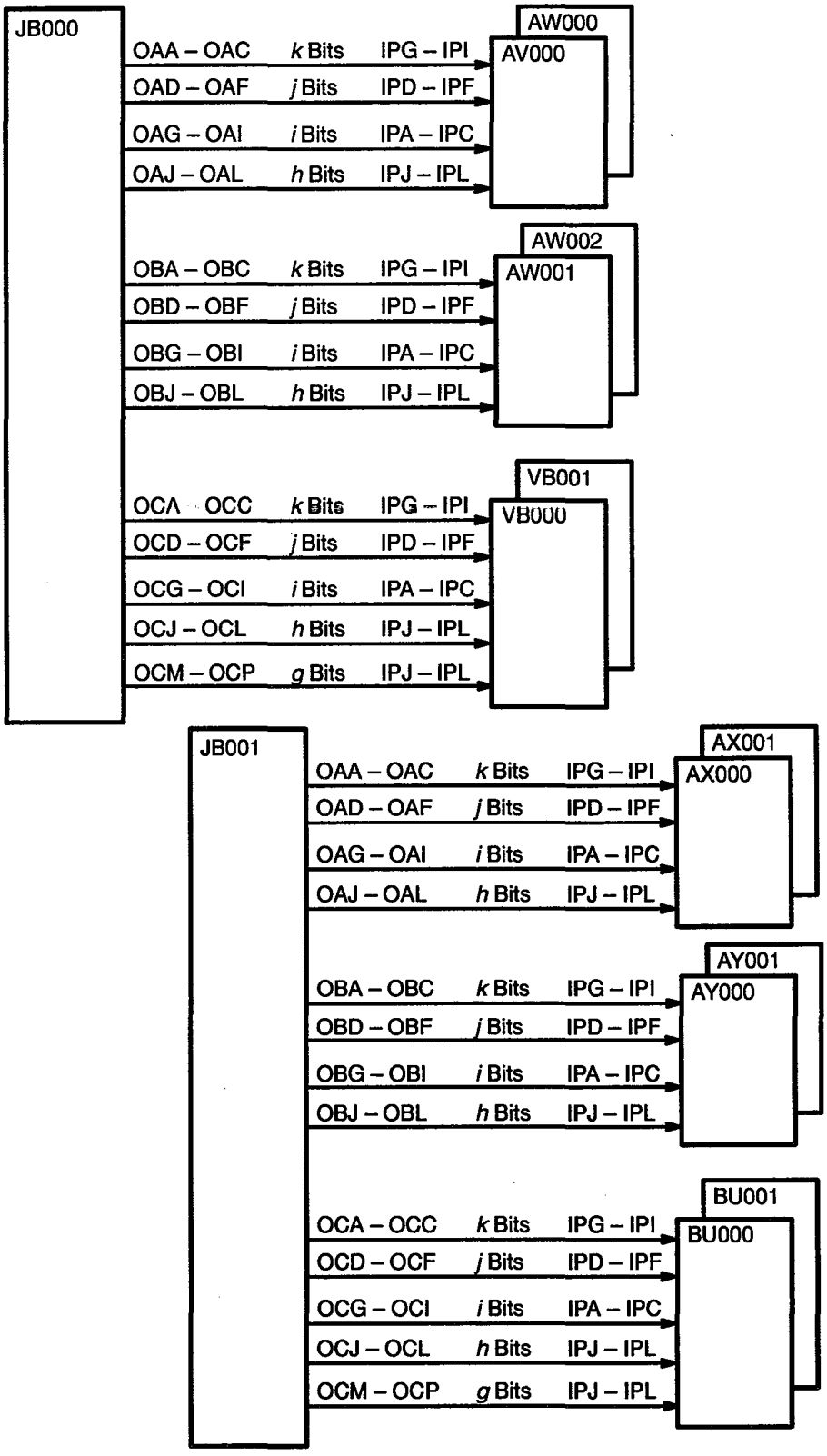


Figure 96. Instruction Data Distribution A/S/B/T/V Registers

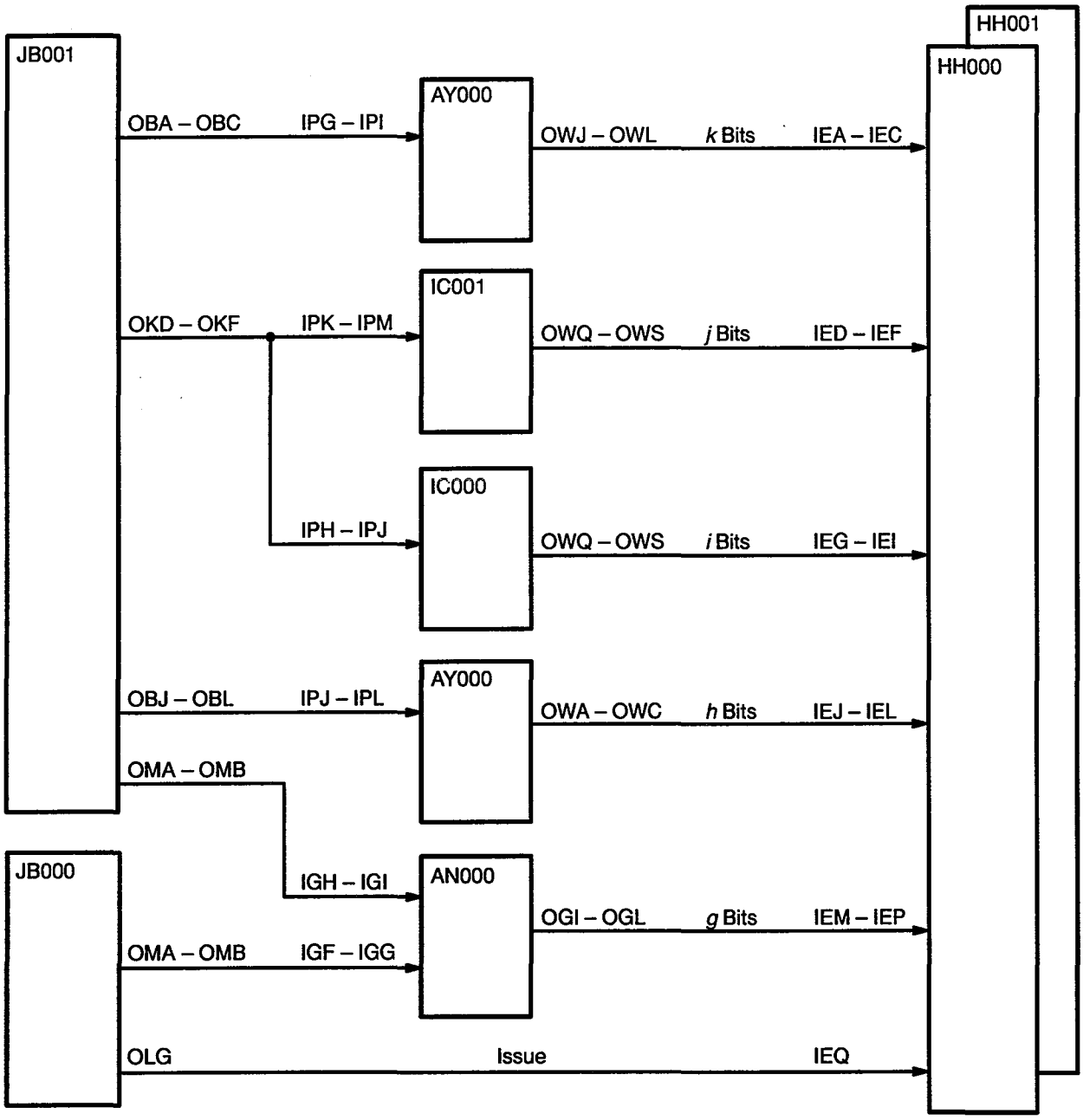


Figure 97. CIP Distribution to HH Options

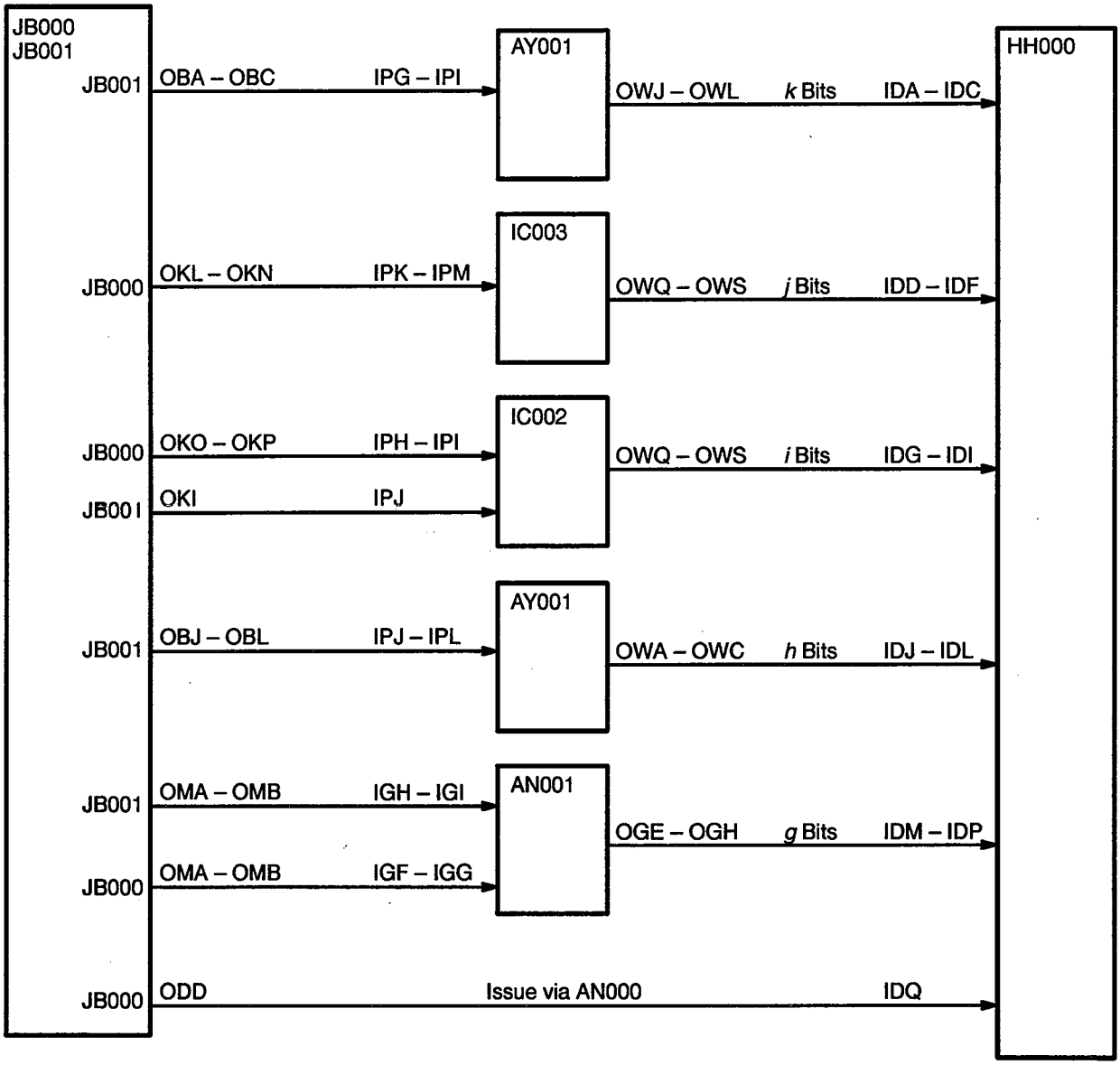


Figure 98. CIP Distribution to HH Option

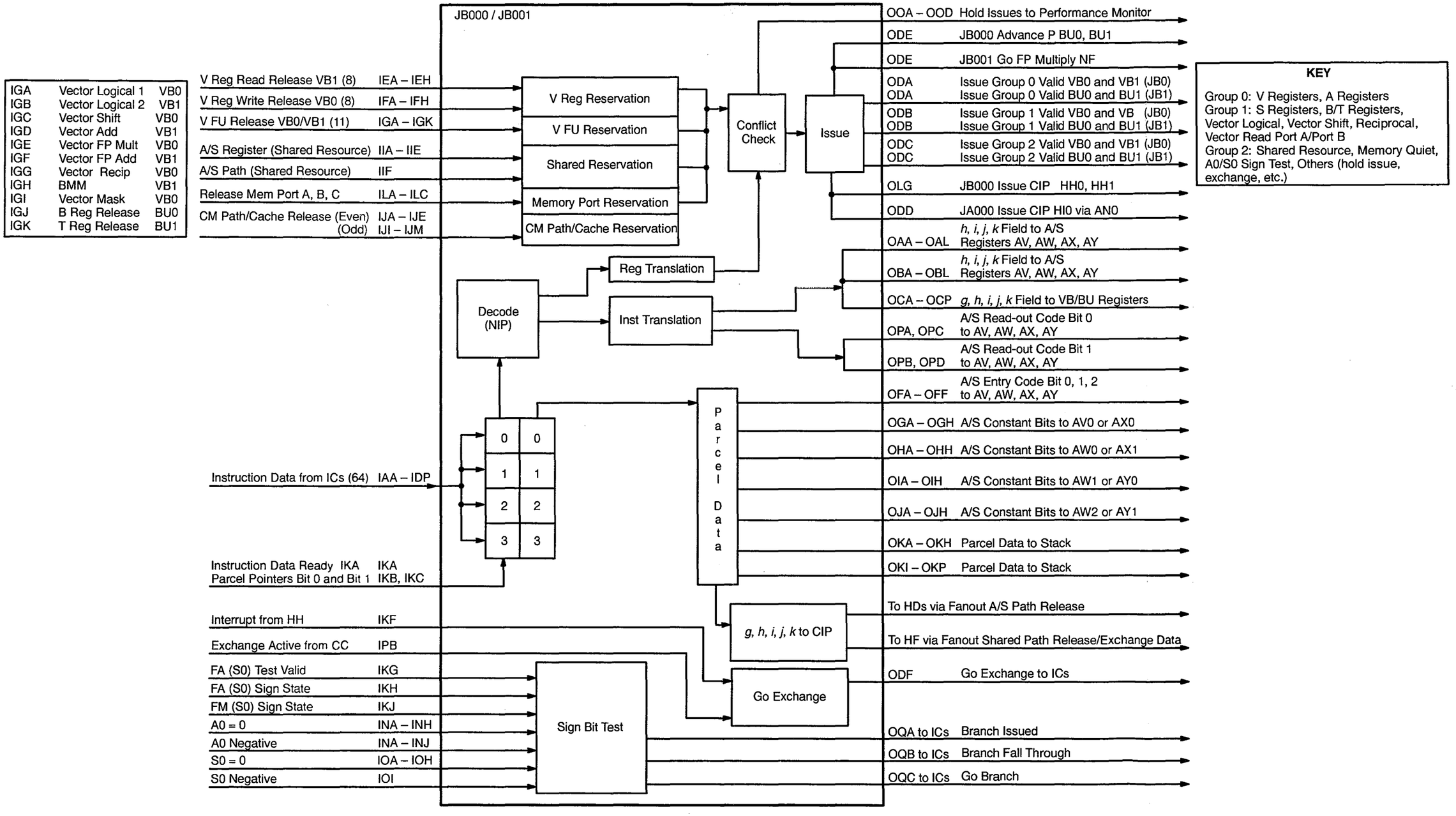


Figure 99. JB Option Block Diagram

EXCHANGE

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_8 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_8 words long
  - Must reside in lower 2 MW of memory
  - Must start on a 40_8 word boundary

Exchange Process

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

In CRAY T90 series systems, a feature in the exchange package 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 can exchange to another user job, or it can exchange to specific areas in the kernel, without first exchanging to the monitor.

The CRAY T90 series system also incorporates another special feature. When an exchange occurs, the CPU that exchanges out retains the cluster number that was initially assigned to it unless the system is operating in C90 mode or unless AutoBCD (automatic broadcast cluster detach) is active. Also, when a CPU is master cleared and then exchanged out, the pending interrupt bits are retained 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, the user is responsible for saving the register values.

Four conditions cause an exchange sequence:

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

SIPI

A CRAY T90 series system does not use a deadstart signal or command. Instead, the system uses Set Interprocessor Interrupt (SIPI) signals from either a 0014*j*1 instruction [send inter-CPU interrupt to CPU (A*j*)], or during an initial deadstart, when a CPU loop controller function of 76_8, issued by the maintenance channel, starts an exchange.

The following sequence lists the events that invoke the Mainframe Maintenance Environment (MME):

1. Set CPU Master Clear.
2. Load data to memory address 0 via the maintenance channel.
3. Issue a loop controller function of 176_8 via the maintenance channel to allow CPU maintenance instructions.
4. Issue a loop controller function of 141_8 via the maintenance channel to allow CPU instruction exchange and halt.

   The exchange package at memory location 0 loads into the CPU registers, and what was in the CPU registers loads to memory starting at location 0. There is no fetch after this exchange.
5. Drop CPU Master Clear via the maintenance channel.
6. Issue the loop controller function of 76_8 via the maintenance channel.

   The dropping of CPU Master Clear works as an enable; the function 76_8 must be present along with the Master Clear signal before the exchange can occur.

7. 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 has been configured. From this point, the initially started CPU can 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 nonmonitor 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 scheme provides a stable environment within monitor mode immediately following an exchange.

Program Exit

Program exit follows the decode of instructions 000000 and 004000. Instruction 000000 is an error exit instruction; 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 (0034*jk*) is in the next instruction parcel (NIP) or entering the current instruction parcel (CIP), the program (P) register will hold the current value until the test and set condition is true. The JB option then waits until the condition is resolved before it advances P. Memory must also be quiet, and all memory writes must be complete.

The processor that is performing the exchange clears 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

load into instruction buffer 0 first. Also, issuing a 0051*jk* instruction clears the buffer valid bits. The 0051*jk* is a maintenance instruction that loads the P register from B*jk* and invalidates the instruction buffers if the CPU is in maintenance mode (MM).

Exchange Package Descriptions

Figure 100 illustrates the exchange package. The exchange parameters are located on two options: HH000 and HH001. HH000 handles bits 0 through 31 for words 0 through 17, and HH001 handles bits 32 through 63 for words 0 through 17.

P Register

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. P register bits –2 through 29 enable the addressing of 1 gigaword of memory.

Modes

Modes – MM, BDM, ESL, SCE, RM0, RM1, BDD word 11, bits 0 through 7. Selectable interrupt modes enable the programmer to choose the conditions under which the active program can be interrupted.

- MM – Monitor mode, word 11, bit 0

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

- 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.

- SCE – Scalar cache enabled, word 11, bit 4

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

- RM0 – Rounding Mode Bit 0, word 11, bit 5

  This is used to determine the rounding mode to be used for floating-point operations.

- RM1 – Rounding Mode Bit 1, word 11, bit 6

  This is used to determine the rounding mode to be used for floating-point operations.

- BDD – Bidirectional memory disable, word 11, bit 7

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

Status

Status (BML, WS, VNU, SBU, SBM) word 12, bit 0 through 7.
Status (NVS, DVS, OVS, UNS, NXS, XIS) word 13, bits 9 through 14.
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.

- BML – Bit matrix loaded, word 12, bit 0

  The BML bit indicates the B^t (B transposed) registers have been successfully loaded by a 1740*j*4 instruction.

- WS – Waiting on semaphore, word 12, bit 1

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

- 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.

- SBU – Status Bit-user mode, word 12, bit 6

  Indicates that the CPU is in user mode.

- SBM – Status Bit-monitor mode, word 12, bit 7

  Indicates that the CPU is in monitor mode.

- NVS – Floating point invalid, word 13, bit 9

  An attempt was made to generate a result that is not a real number. Invalid is signaled in any of the following cases:

  - An input operand is an sNAN
  - Addition or subtraction of infinites
  - Multiplication of 0 by infinity
  - Division of 0 by 0 or infinity by infinity
  - Division of a finite normal numerator by 0
  - Square root of a negative number
  - Signed compare where one or both inputs are NaNs

- DVS–Floating point divide by zero, word 13, bit 10

- OVS–Floating point overflow, word 13, bit 11

  A result larger than the greatest representable number was generated. Infinity (03777 0000000000000000) is returned.

- UNS–Floating point underflow, word 13, bit 12

  A result smaller than the least representable number was generated. Zero (00000 0000000000000000) with the sign bit is returned.

- NXS–Floating point not exact, word 13, bit 13

  A result was generated that would be different if all possible significant bits were returned. Inexact is also signaled on both overflow and underflow, but not if the returned result is exactly 0.

  - 1 / 3 returns 0.33333.....3 and signals Inexact
  - 0.5 / 2 returns .25 all bits returned.
  - A floating-point unit received an operand of infinity or NaN. This is a Cray Research feature not an IEEE standard.

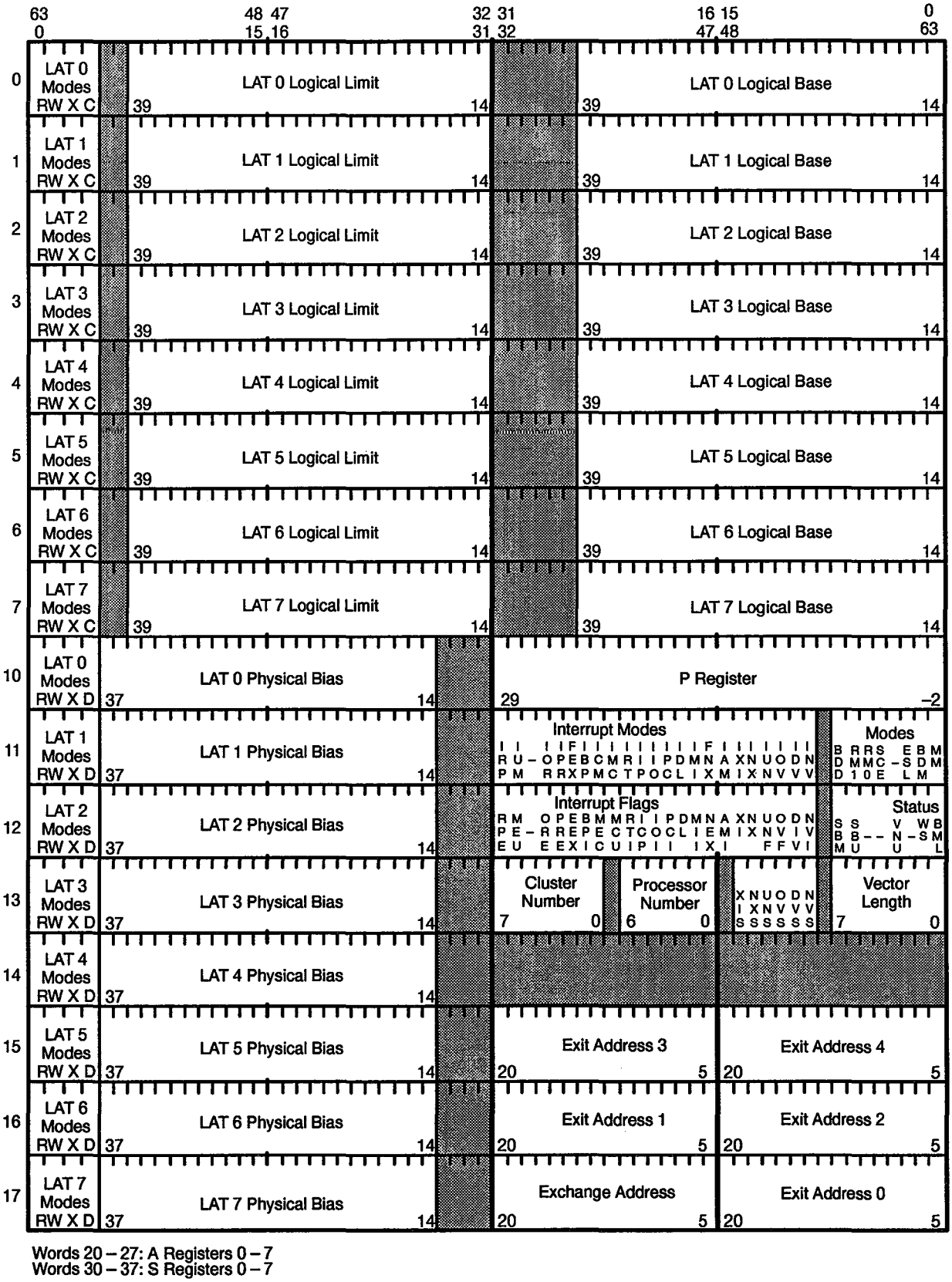


Figure 100. Exchange Package

Interrupt Flags

Interrupt modes, word 11, bits 9 through 31. Refer to Table 48 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 48. 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	–	Not Used
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*jk*≠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
11	14	IXI	Interrupt on floating-point exceptional input
11	13	INX	Interrupt on floating-point not exact
11	12	IUN	Interrupt on floating-point underflow
11	11	IOV	Interrupt on floating-point overflow
11	10	IDV	Interrupt on floating-point divide by zero
11	9	INV	Interrupt on floating-point invalid

Refer to Table 49 for a list of the bit assignments for the interrupt flags field in the exchange package.

Table 49. Flag Register Bit Assignments

Word	Binary Exponent	Acronym	Name
12	31	RPE	Register Parity Error
12	30	MEU	Uncorrectable Memory Error
12	29	–	Not Used
12	28	ORE	Operand Range Error
12	27	PRE	Program Range Error
12	26	FFX	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)†
12	19	PCI	Programmable Clock Interrupt
12	18	DL	Deadlock Interrupt
12	17	MII	001*jk*≠0 or 033 Instruction Interrupt (if IMI and not MM)
12	16	NEX	Normal Exit (004 issued)
12	15	AMI	Address Multiply Interrupt
12	14	XI	Floating-point exceptional input interrupt
12	13	NX	Floating-point not exact interrupt
12	12	UNF	Floating-point underflow interrupt
12	11	OVF	Floating-point overflow interrupt
12	10	DVI	Floating-point divide by zero interrupt
12	9	NVI	Floating-point invalid interrupt

† SIE = System I/O interrupt enabled.

Vector Length

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_8 vector elements are used within the vector register.

Exchange Address

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

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.

Cluster Number

CLN – cluster number, word 13, bits 24 through 31. The CLN contains an 8-bit field. There may be up to 36_8 clusters in the system, depending on the system configuration.

Processor Number

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.

Logical Address Translation

LAT – Logical address translation, words 0 through 17. Refer to the exchange package diagram for bit layouts. Each LAT has four associated fields; Table 50 identifies those fields.

Table 50. 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 can reference the same library routine in memory while keeping their local code private.

SCALAR CACHE

Each CPU has a scalar data cache. The data cache accelerates common memory data access for address register and scalar register read requests. Only address and scalar registers can access the data 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, which provides 1,024 words of data in the cache. Figure 104 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.

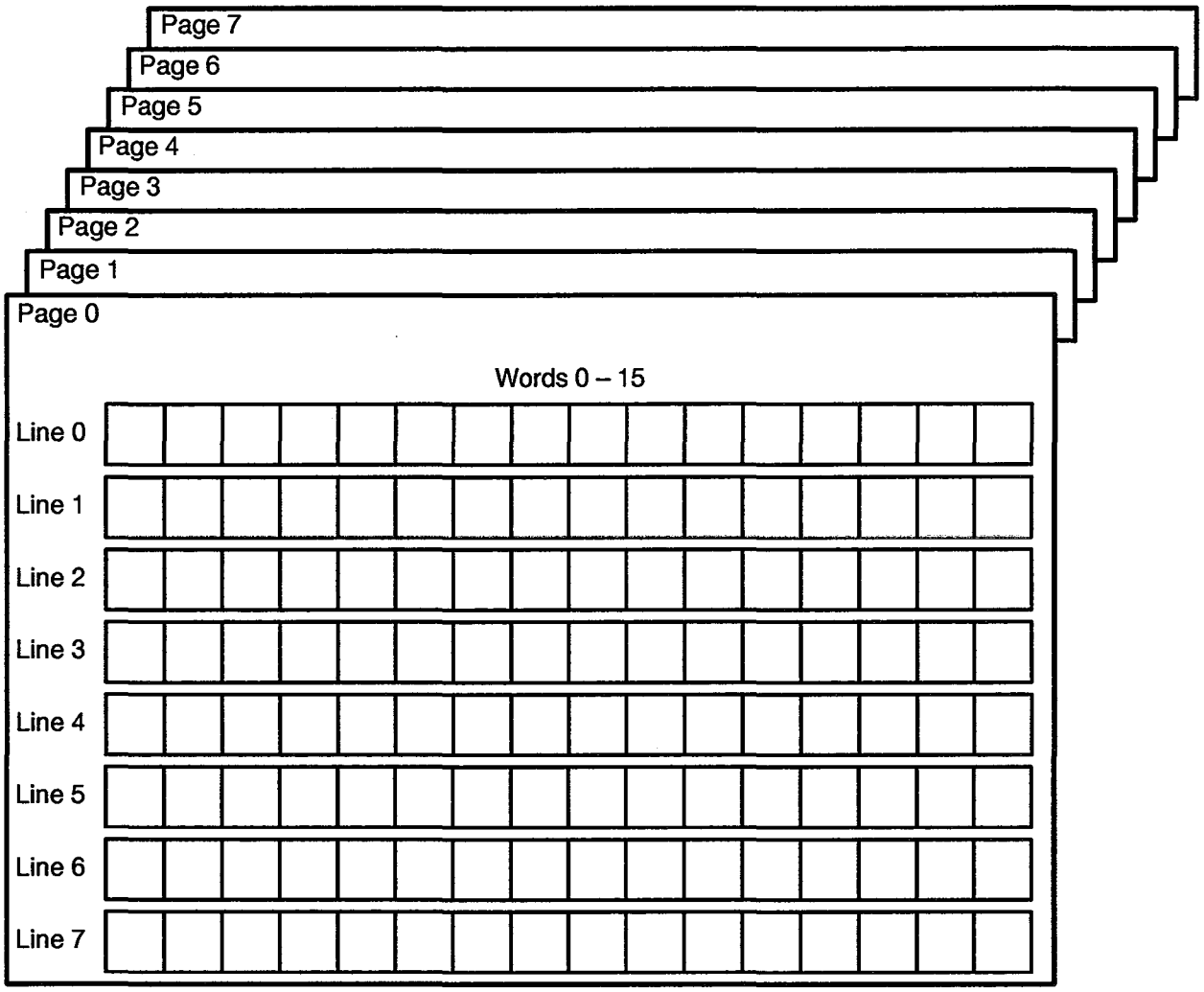


Figure 104. Cache Layout

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 if the page is not valid. When this occurs, 16 sequential words are requested from memory, and the line is set valid.

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) in each central processing unit (CPU). The RTC is synchronized when a CPU issues a 0014*j*0 instruction. The 0014*j*0 instruction causes all CPUs in the same cluster to be loaded with the contents of S*j*.

The RTC is located on two HH options, each of which handles 32 bits. The HH000 option handles bits 0 through 31; the HH001 option handles bits 32 through 63.

HH000l detects a carry from the RTC, at a count of 37777777776 during normal operation and increments the upper bits during the next clock period. HH000 suppresses any toggles.

The RTC is incremented each clock period. The RTC enables clock-period timing of program execution. When the machine is deadstarted, all RTCs must be loaded in order to synchronize all the CPUs. Otherwise, each CPU will have a different RTC value.

The 0014*j*0 instruction writes to the RTC by sending a copy of the S*j* register from the CPU issuing the instruction to all RTC registers through the issue paths of the shared registers. The 072*i*00 instruction reads the RTC register of the CPU that issued the 072*i*00 instruction and copies the content into the scalar registers.

Refer to Figure 101 for an RTC and programmable clock interrupt (PCI) block diagram.

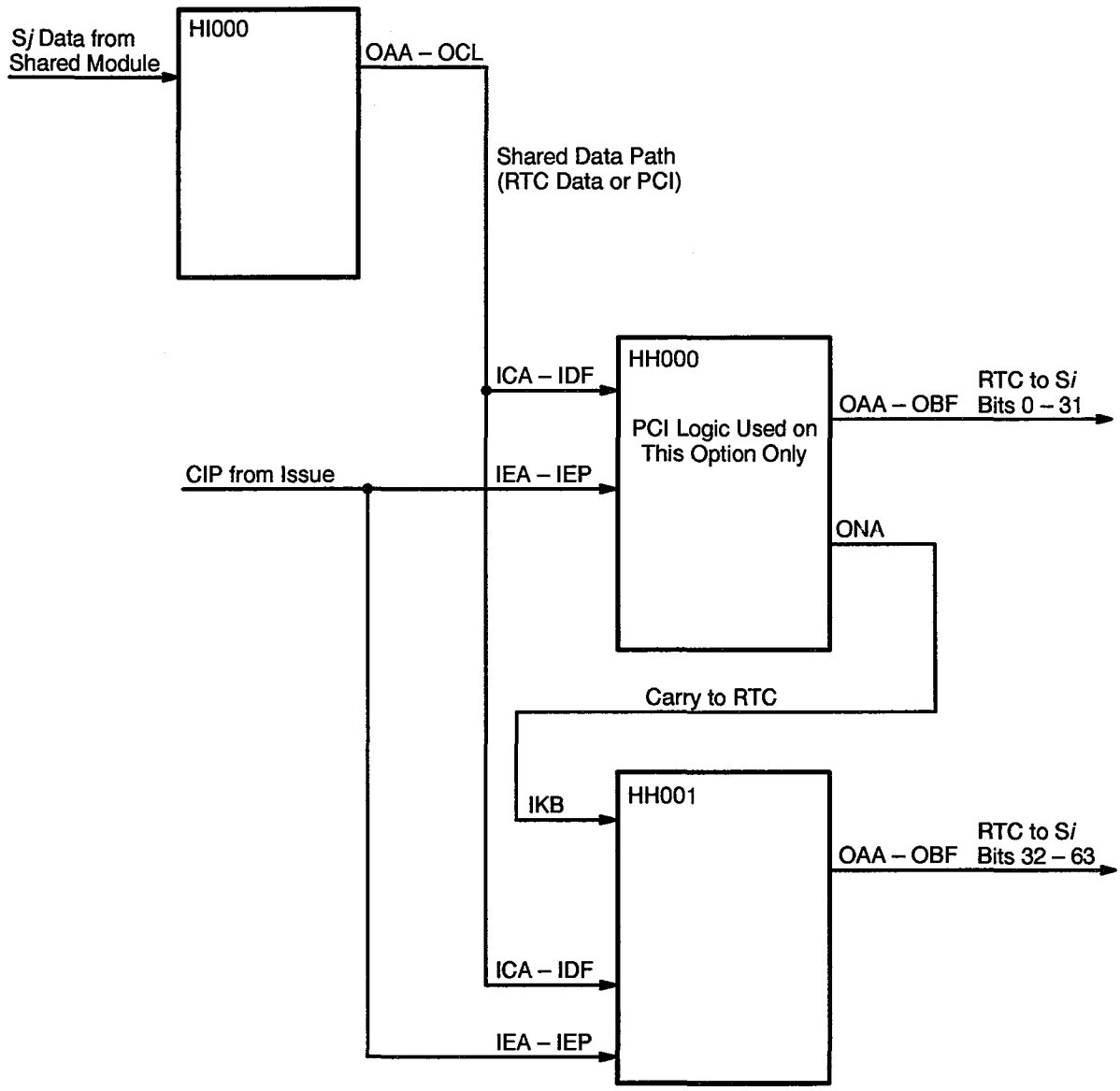


Figure 101. RTC and PCI Block Diagram

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 0014*j*4 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 a 001405 instruction clears it. 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 HI option, which contains the exchange parameters.

RTC and PC Instructions

Refer to Table 51 for a list of the RTC and PC instructions.

Table 51. RTC and PC Instructions

Instruction	CAL	Description
0014*j*0 †	RT S*j*	Enter RTC register with S*j*
072*i*00	S*i* RT	Transmit RTC to S*i*
0014*j*4 †	PCI S*j*	Transmit S*j* to programmable clock
001405 †	CCI	Clear PCI request
001406 †	ECI	Enable PCI request
001407 †	DCI	Disable PCI request

† 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; each counter is 48 bits wide. Table 52 shows which event each counter monitors. Each counter increments each time a particular event occurs in the CPU while the CPU is not in monitor mode (IMI bit is not set). The counters related to memory references may increment as many as eight times per clock period (CP). Counters related to vector operations increment by the value in the vector length register at the time the instruction issues.

Table 52. 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, 073*i*20, 073*i*30	+1
23	B/T memory instructions	034 – 037	+1
24	Scalar instructions	040 – 043, 071 – 077 except 073*i*20, 073*i*30	+1
25	Scalar integer instructions	044 – 061, 070*ij*6	+1
26	Scalar floating-point instructions	062 – 070	+1
27	S/A memory instructions	10x – 13x	+1
	Number of operations:		
30	Vector logical	070*ij*1, 140 – 147, 1740*i*4 – 1740*i*6, 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, 161, 165, 166	+VL
34	Vector floating-point add/compare/converts	167 – 173	+VL
35	Vector floating-point divide/square root	162, 163, 174x*j*0	+VL
36	Vector memory reads	176	+VL
37	Vector memory writes	177	+VL

Performance Monitor Instructions

Table 53 lists all the instructions associated with the performance monitor.

Table 53. Performance Monitor Instructions

Instruction	CAL	Description
001500		Clear all performance counters
073*ij*1	S*i* SR*j*	Transmit (SR*j*) to S*i* (monitor mode only for *j* = 2 – 7)
073*i*05	SR0 S*i*	Transmit (S*i*) bits 48 – 52 to SR0
073*i*25	SR2 S*i*	Advance performance monitor pointer
073*i*75	SR7 S*i*	Transmit (S*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 073*i*21 and 073*i*31 instructions read the performance monitor. Each instruction reads half of the counters at a time, which requires that two instructions be issued to read all the counters. The 48 bits of the counter read are stored in the S*i* register. When the 073*i*21 instruction is issued, counters 0 through 17 are sent to S*i*. The 073*i*31 instruction, when issued, reads counters 20 through 37 and sends the bits to S*i*.

The system hardware requires an interval of at least 3 clock periods between 073*ix*1 instructions, and 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 indeterminable corruption of performance monitor data occurs.

Performance Monitor Block Diagram

Refer to Figure 102 for the performance monitor block diagram. The performance monitor is composed of the HI000, HH000, and HH001 options. The HI000 option contains the lower bits (0 through 31) and the HH000 and HH001 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 increment 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 HH and HI options. The status register is not part of the exchange package in CRAY T90 series systems. Figure 103 shows the status register format and bit assignments of each register. The status registers are read by the 073*ij*1 instruction.

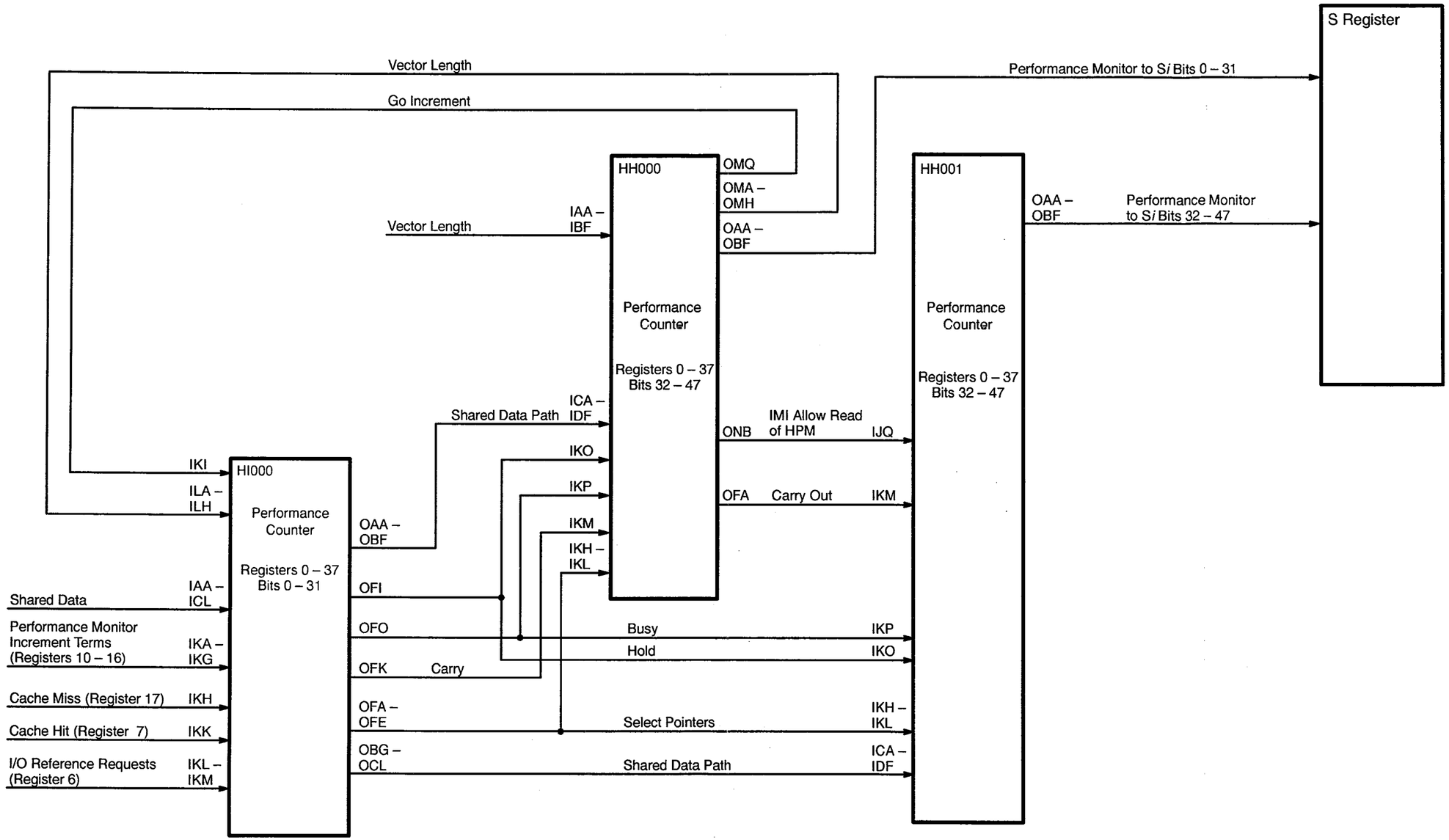
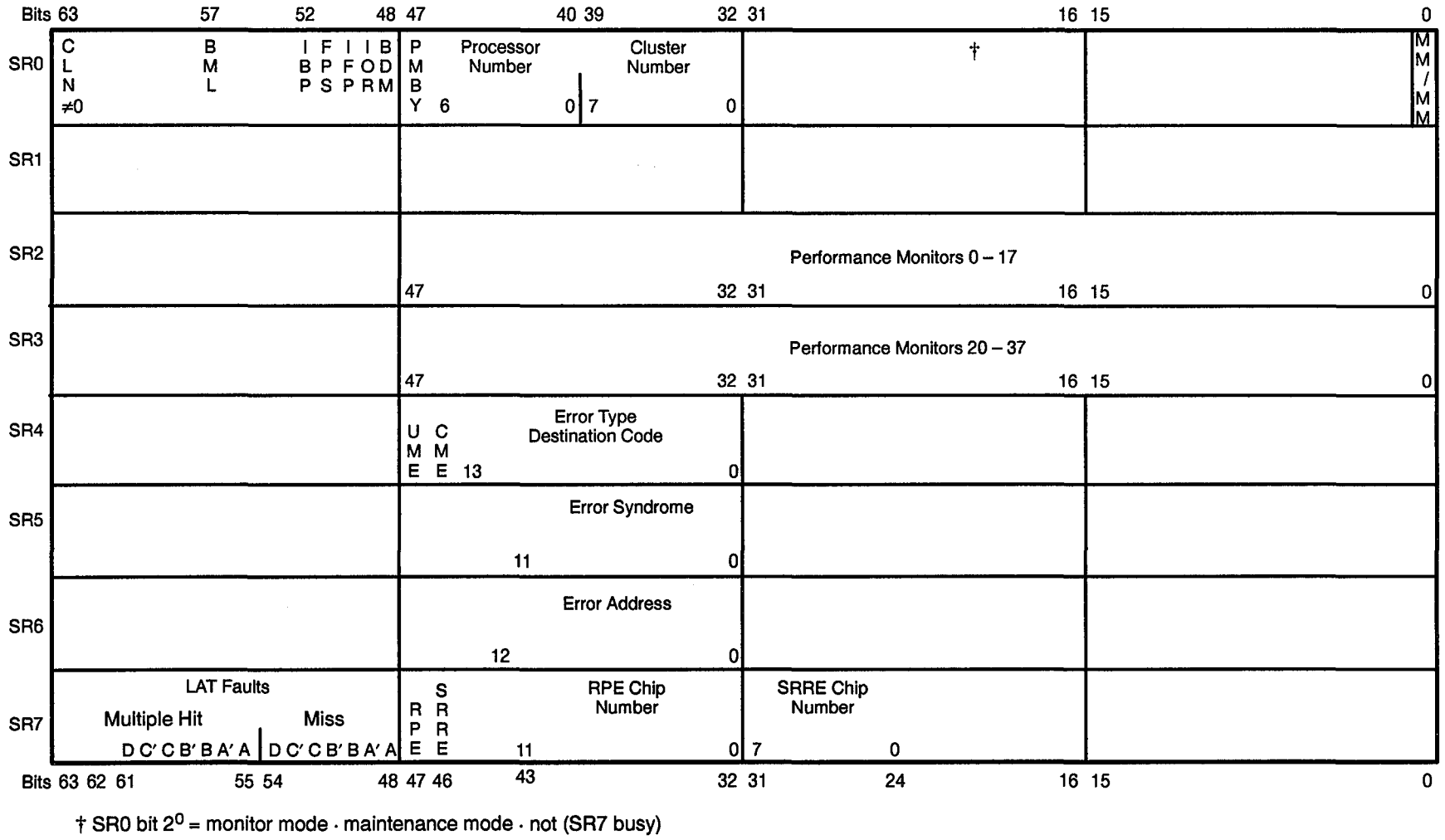


Figure 102. Performance Monitor Block Diagram



† SR0 bit 2^0 = monitor mode · maintenance mode · not (SR7 busy)

Figure 103. Status Registers

The eight status registers are further defined in Table 54 through Table 57.

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

Table 54. Status Register (SR0)

Bits	Name	Description
63	CLN≠0	Cluster number not equal to zero
57	BML	Bit matrix loaded
47	PMBY	Performance monitor busy
40 through 46	PN	Processor number
32 through 39	CLN	Cluster number
31	SMB †	Interrupt on floating-point error
30	SMU †	Interrupt on operand range error
20	IBP †	Interrupt on breakpoint
19	IOR †	Interrupt on operand range error mode
18	BDM †	Bidirectional memory mode
17	SCE †	Scalar cache enabled
16	XIS †	Floating-point exceptional input
15	NXS †	Floating-point not exact
14	UNS †	Floating-point underflow
13	OVS †	Floating-point overflow
12	DVS †	Floating-point divide by zero
11	NVS †	Floating-point invalid
9	IXI †	Interrupt on floating-point exceptional input
8	INX †	Interrupt on floating-point not exact
7	IUN †	Interrupt on floating-point underflow
6	IOV †	Interrupt on floating-point overflow
5	IDV †	Interrupt on floating-point divide by zero
4	INV †	Interrupt on floating-point invalid
2	RM1 †	Floating-point round mode bit 1
1	RM0 †	Floating-point round mode bit 0

† 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 55. 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 56 is a decode of these destination bits.

Table 55. Status Register 4 (SR4)

Bits	Name	Description
47	UME	Uncorrectable memory error
46	CME	Correctable memory error
32 through 45	CODE	Destination code (refer to Table 56)

Table 56. 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 57. Status Register 7 (SR7) 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

Cache Addressing

Figure 105 shows how memory addresses are used to determine a cache hit or miss.

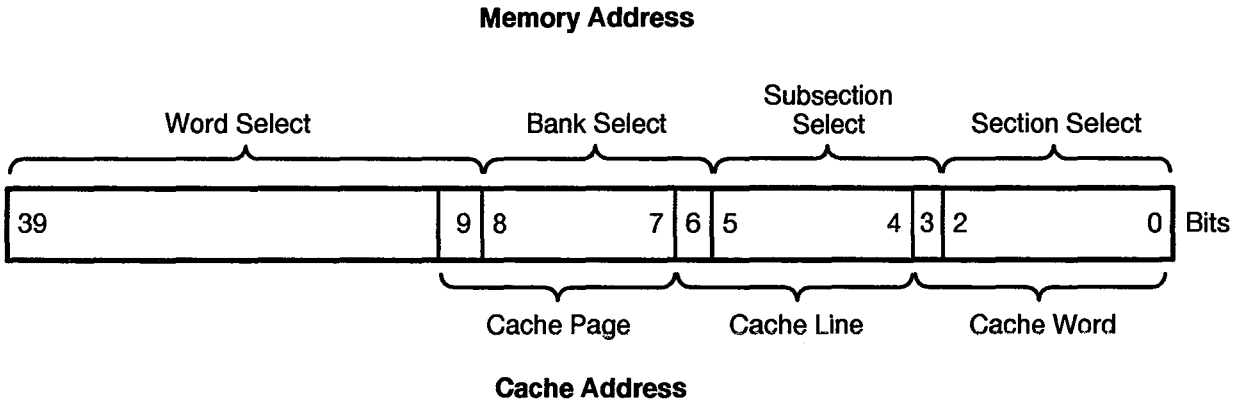


Figure 105. Memory Addresses

Potential Cache Problems

Because no communication occurs between caches in different CPUs, 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 writes 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 *thrashing* memory with a stride value of 128. A stride of 128 uses 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 is avoided by redesigning user code.