CRAY-2 Central Processor

The CRAY-2 central processor is an evolutionary step from the design of the first Cray Research machine toward greater parallel computing in a monoprocessor mode. The new machine has somewhat greater hardware speed (eight nanoseconds per clock period vs 12.5 nanoseconds) and much wider bandwidth throughout. The net effect of these changes is a factor of four speed improvement for vector-oriented scientific computation. The data processing structure is simplified over the earlier design and is much less sensitive to program quality. That is, a poorly written program will run better on a CRAY-2 machine than the equivalent program would on a CRAY-1 machine. This is the case because the timing of instruction issue and execution is built into the hardware of the new machine and careful program coding is not as necessary.

The CRAY-2 architecture began with a new assembly language (CPAL) and the hardware was then designed to implement the concepts in this software approach. The new language is a medium level language and as such provides a good vehicle for compatible execution of CRAY-2 programs on a CRAY-1 machine. This language does not have a one-for-one relationship between source language statements and binary machine instructions. It is hardware oriented in the sense that the resources of the machine are specifically identified in a qualitative sense in the source language statements. Assignment of resources in a quantitative sense is done partially by the compiler and partially by the hardware at execution time.

The functional unit operations of the CRAY-2 hardware are essentially identical to those of the CRAY-1 hardware. The word length, floating point format, arithmetic modes, and general computation philosophy are all the same. The differences appear as a result of the need for greater bandwidth. More operand registers are required for both the scalar and the vector sections of the machine. The instruction format is changed from a three address to a one address form to accommodate these expanded parameters. One result of this expanded operating field is an increase in program code over the CRAY-1 machine equivalent code of 20 to 30 percent. The larger volume of code is executed faster by issuing more than one instruction per clock period. Maximum issue rate is four instructions per clock period or two nanoseconds per one address instruction. Execution of the issued instructions is allowed to proceed out of time sequence over large sections of the code. This is accomplished by operating register reservations with unlimited read/write pattern structure. Bandwidth in the arithmetic sections of the machine is obtained by making the depth of the hardware invisible to the programer and allowing the hardware to convert an apparently simple serial program into a wide bandwidth parallel one.

Internal Data Communication Paths

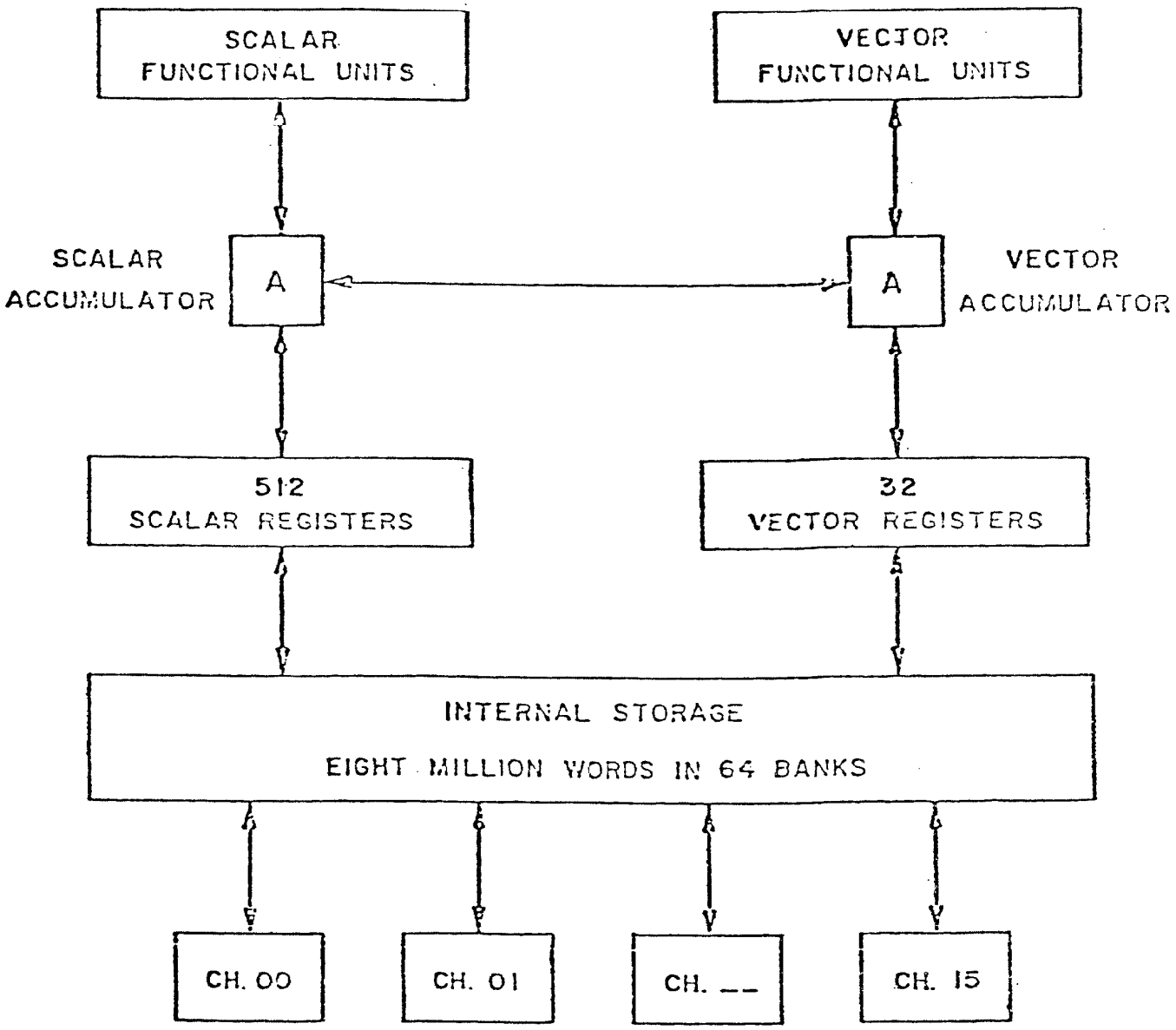
The internal data communication paths are illustrated in figure 1 to provide an overview of the central processor internal structure. This structure is dominated by the internal memory which consists of eight million words of 64 bits each. The memory is constructed of 4096 bit bipolar integrated circuit chips which are arranged in 64 banks with interleaved addressing. This memory represents the bulk of the hardware and contains a large portion of the internal data communication paths. Each of the 64 banks are individually addressable from a number of sources. This allows many unrelated storage references to be going on concurrently.

The input and output channels for the central processor each have their own hardware for internal storage access control. Hardware registers in the access control define the beginning address and limit address for a buffer area. This buffer area may be in any part of the memory and of any size. These parameters are established by the central processor monitor program when an input or output operation is initiated.

Internal computation in the central processor is divided into two classes each with associated hardware. The scalar computation section utilizes 512 operand registers each capable of holding a single 64 bit data word. Computation is performed using one address instructions with continuity through a single 64 bit scalar accumulator. Scalar functional units are accessible through the accumulator which is normally the source of one operand and the destination of the result.

Vector computation is performed in a separate section of the machine and uses 32 operand registers each capable of holding 64 data words of 64 bits each. Vector lengths less than 64 elements use a portion of a vector operand register. Vector lengths greater than 64 elements are segmented as appropriate to fit in these registers. Vector computation is performed using one address instructions with continuity through a single 64 element vector accumulator. This accumulator is reentrant so that multi-address vector sequences may be chained together for simultaneous execution. The vector functional units are all segmented so that operands may enter the units each machine clock period. The vector accumulator and vector functional units work together synchronously on streams of data which flow uninterrupted through the network. Four floating multiply units and four floating add units provide a potential for one billion floating point operations per second.

CENTRAL PROCESSOR DATA PATHS



16 INPUT/OUTPUT DATA CHANNELS

Instruction Format

Program code is stored in the machine internal memory at execution time and is read into an instruction issuing mechanism as the individual instructions in the program are required. The machine internal memory is organized in 64 bit words and such a 64 bit section of program code is called a 'program word'. Each program word is further divided into quarters called 'instruction parcels'. The 16 bit instruction parcels are the basic program units. These consist of two designators with complementary attributes which coexist in the configuration shown below.

7 bits 9 bits

The F designator is the instruction function code and specifies which of the instructions in the machine repertoire is intended for execution by this instruction parcel. The D designator has several uses, depending on the instruction, but in general specifies were in the machine resources the function of the F designator is to be performed. The D designator may be thought of as a displacement specification. It is used to point to a specific scalar operand register where one is required, or to a specific vector register. It is used to indicate how far forward, or backward, a program branch instruction intends to move the program address. It is used to increment or decrement an index or an operand address where a small displacement from a base value is required.

There are no address length constants in the body of the program code. All such parameters are held in tables associated with the program code but grouped generally at the end of program subsections and referenced through a base address and a displacement much as a data array. This approach makes the program code independent of location in the internal memory at execution time.

Instruction Processing

Program code is read from the central processor internal memory and executed via a number of steps as illustrated in figure 2. Program words are read sequentially from the internal memory to fill a 64 word instruction buffer. These buffers hold a block of program code which begins and ends at 100 (octal) word boundaries in the internal storage. There are four such buffers which hold a 100 (octal) word block each. The four buffer fields need not be contiguous.

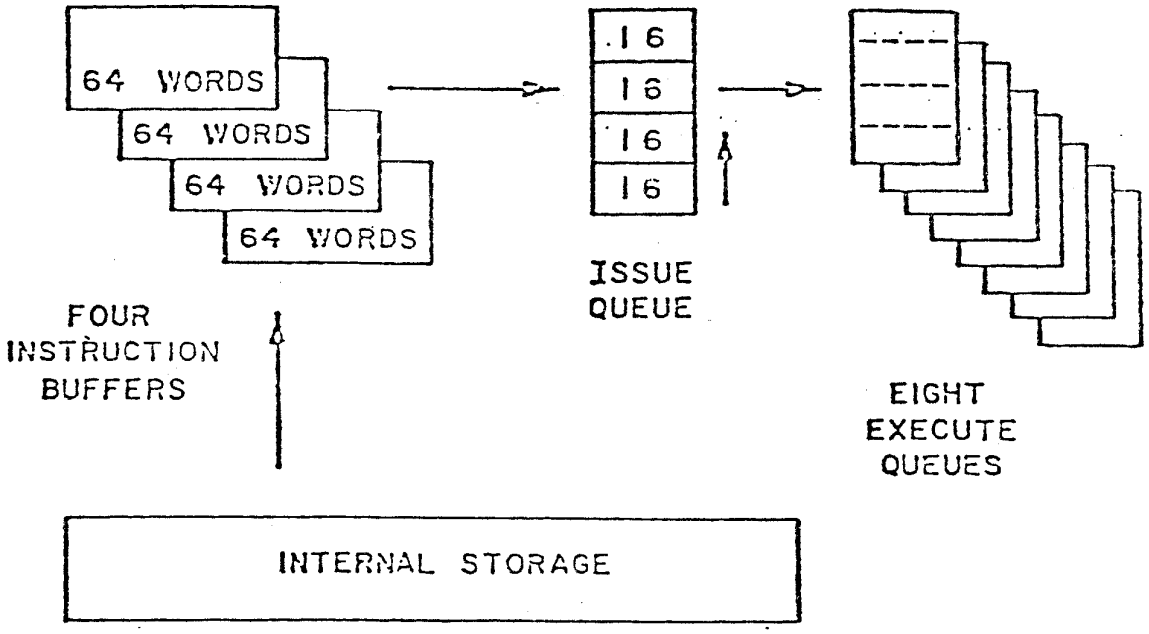
An instruction buffer load sequence begins when the executing program requires an instruction word which is not currently in one of the four buffers. A series of storage references is initiated beginning at the storage location required by the executing program. This storage address need not be at a 100 (octal) word boundary and in that case the buffer would begin filling at an arbitrary internal location. The filling process continues to the end of the buffer and then fills the beginning of the buffer to the original address requested. Program execution may proceed as soon as the first word arrives at the buffer and will continue reading code from that buffer until a program word is required which is out of the range of this block. At that time the next sequential instruction buffer is cleared of its current program code and an instruction buffer load sequence is initiated for that buffer.

A program word is moved from an instruction buffer to the issue queue when that word is required for execution. This is the last point in the instruction processing sequence where the 64 bit program word is treated as an entity. The issue queue treats the program word as four independent instruction parcels. An instruction parcel leaves this queue when that instruction issues.

Instruction issue refers to the process of committing a parcel of program code to execution. When one instruction parcel in a word has been issued for execution the remainder of the parcels in that word must also be issued before the program sequence can be interrupted. The parcels in a program word are ordered from left to right. All four parcels may issue in the same clock period if the proper conditions exist. If some of the parcels issue and the remainder do not, the remaining parcels shift up in the issue queue and the low order positions are zero filled. When all four parcels have been issued the next program word is read from the instruction buffer.

Issued instructions move to an execute queue. There are eight execute queues each with a four instruction parcel capacity. The compiler tends to adjust code so that full program words are assigned to execute queues. This need not be the case and a part of one word and the beginning of the next may be assembled in an execute queue. This process would require more than one clock period as only one execute queue can be entered in a given clock period.

INSTRUCTION PROCESSING



Instruction Issue

Instruction issue refers to the process of committing a parcel of program code to execution. When one parcel in a word has been issued for execution the remainder of the parcels in that word must also be issued before the program sequence can be interrupted. Program words are read from the program buffer into an issue queue. This queue holds the four parcels of one program word. One or more of these parcels may issue from the queue in the next clock period. When all four parcels have been issued the program word is replaced with a new word from the program buffer.

Issued instructions move to an execute queue. There are eight execute queues associated with eight scalar accumulator networks. A particular execute queue is normally first entered with an instruction which clears the accumulator and enters new data. Following instructions in the program sequence are stacked in this execute queue until the queue is filled or a new clear and enter accumulator instruction appears. If the queue is filled, the next sequential execute queue is reserved to continue the program sequence. A clear and enter accumulator instruction always begins a new execute queue and that queue need not be in sequence. The accumulator networks and associated execute queues are assigned in sequence until a busy network is encountered. That sequential execute queue is then skipped and the next sequential execute queue tested for busy in the following clock period.

Instructions which specify a B or L parameter are interpreted at issue time. That is, the B register value is read at issue time and an equivalent instruction with a direct register designation is substituted as the instruction is moved to the execute queue. The L register value is captured and held at the execute queue when the first instruction which uses the L parameter issues to that execute queue. There is only one position at the queue for this parameter. If a second instruction in the program sequence uses a different L parameter the filling of the next sequential execute queue is initiated.

Instructions which alter the B or L parameters are executed directly from the issue queue. The new values for these parameters are then available in the following clock period. Instructions which involve a program branch are executed directly from the issue queue. If the branch is conditional and is based on the L parameter it may proceed immediately. If the branch is conditional and is based on the W register it must wait until the W register is free.

Subset A or B instructions which include an advance A flag issue to the execute queue with the flag attached. The associated accumulator content is increased by one count as that instruction is released for execution from the execute queue.

Instruction Execution

A central processor instruction is interpreted and the necessary control signals generated for execution at the time the instruction parcel is released from an execute queue. These control signals must be coordinated with those for the other instructions in the same computational sequence. The sequences tend to be of three or four parcel length. The most common examples are: Load, Add, Store; and Load, Add, Memory reference. The CPAL statement formats and hardware implementation both reinforce these patterns.

Instructions are stacked in an execute queue for the execution of a single computational sequence. A particular execute queue is normally first entered with an instruction which clears the scalar accumulator and enters new data from a scalar operand register. Following instructions in the sequence are stacked in this queue until the queue is full or until a new clear and enter instruction is encountered. The content of the execute queue normally corresponds with one CPAL statement. This sequence is then executed by the hardware control mechanism as an integrated unit with optimal timing. The control signals generated by execution of the instructions in one execute queue can be quite independent of the activity in the other seven queues. Scalar and vector operand register reservations are made at the time the instruction parcels leave the issue queue. These reservations insure that the execution of the sequences do not get out of time sequence when that sequence is important. The only other interaction between execute queue sequence controls is the availability of the resources in the sense of register bank and functional unit conflicts.

A clear and enter scalar accumulator instruction always issues to a new execute queue. Execute queues are examined in sequence for availability. If the next queue in sequence is busy the instructions are held in the issue queue and the next execute queue is examined in the following clock period. Consecutive computational sequences may therefore be executed from queues which are not adjacent. A computational sequence which contains more than four instruction parcels must use two execute queues chained together. In this case the execute queues must be adjacent because of hardware limitations in chaining the sequences. Long sequences may therefore be delayed in issue because of availability of execute queues.

Instructions which specify a B or L parameter are interpreted at issue time. That is, an instruction which specifies a B parameter causes the content of the B register to be captured and transmitted to the execute queue at the time the instruction parcel leaves the issue queue. The B register content in this case assumes the role normally filled by the D-designator and causes the associated operand register reference to be indirect. Each execute queue has one storage position for an L parameter. This position is filled as an instruction which uses the L parameter issues. A second instruction in the same sequence with a different L value requires that the next execute queue be chained for continuation of the sequence.

Program Exit Stack

The program exit stack is a mechanism for retaining the program return addresses in subroutine calls until they are needed. The stack consists of 16 registers, each of 24-bit length. A four-bit register, E, serves as the stack pointer to indicate where in the stack the current subroutine exit address is stored.

The E register is cleared when all subroutine calls have been satisfied. Each execution of a return jump instruction advances the E register content by one count and enters the calling program return address in the newly pointed register. Each execution of an exit instruction (00005) in the called program reads the content of the pointed register for the return address and then reduces the E register content by one count.

The E register may be read and altered by the object program so that an error exit from a called subroutine may abort the system of nested return addresses and proceed to an arbitrary level in the stack. The actual return addresses in the program exit stack cannot be altered by the object program.

The execution of a return jump instruction with E = 15 (octal) sets the exit stack overflow flag in the M register. This indicates that the object program has called nested subroutines to a depth where the hardware mechanism is approaching a limit. The return jump instruction is completed in a normal manner, storing the return address in program exit stack register 16. The object program is then interrupted for monitor program service, and the entrance address for the called subroutine is stored in program exit stack register 17 (see interrupt sequence). The monitor program then reorganizes the program exit stack to permit continuing subroutine calls.

The execution of an exit instruction with E = 0 sets the exit stack underflow flag in the M register. This indicates that an object program has returned through all of the return addresses currently nested in the program exit stack hardware, and monitor program intervention is necessary to replenish the stack. The monitor program entrance address is read from register zero in the program exit stack, and execution of the monitor program reorganizes the stack. The monitor program exits with the next higher level subroutine return address in program exit stack register 17.

There are 16 (octal) usable positions in the program exit stack for return addresses. Register zero is reserved for the monitor program entrance address, and register 17 is reserved for the interrupt return address.

Trapping Small Loops

Special hardware is provided for trapping small loops of instructions in the issue queue. The condition required for the trapping mode to apply is an 023000 instruction in the second, third, or fourth parcel of a program word. This instruction is a branch on L ≠ 0, with the branch destination the beginning of the current program word. This package of code is essentially a repeat L times sequence. The package of code within the loop is issued as a unit with suitable substitutions of register designation. Each issue modifies the B and L register values as appropriate and issues the instructions to an execute queue. The process may be interrupted at any point since the continuity is in the B and L registers. An example of such a loop is the following in CPAL.

```
TAA   A@ = (A).
      A = A + 1.
      B = B + 1.
      L = L - 1.
      P = TAA, L ≠ 0.
```

This sequence loads a block of storage data into a series of scalar registers. The first four lines of source language would be compiled into one parcel. This would be compiled as instruction 001057. The 023000 instruction would appear in the second parcel of the issue queue. The first parcel in the issue queue would issue repeatedly to an execute queue. Each issued parcel would have an advanced register designation, and the register would be reserved at issue time. An advance A flag would be issued along with each instruction. The B parameter would be advanced and the L parameter reduced as each instruction issues. This process would continue until L = 0. The following parcel in the program word would then be interpreted.

M Register

The M register is a collection of mode and error flags which is used by the monitor program in processing a monitor call. This register provides the object program parameters necessary for the monitor program to determine the cause of interruption. The flags are listed below with the bit numbers for the associated scalar accumulator bit positions.

53) Monitor mode flag
54) Floating point mode flag
55) Monitor call flag
56) I/O channel request flag
57) FP range overflow flag
58) Program bounds overflow flag
59) Operand bounds overflow flag
60) Exit stack overflow flag
61) Exit stack underflow flag
62) Object program zero flag
63) Object program sign flag

Monitor Mode Flag -

The monitor mode flag is set by the interrupt sequence at the time that object program execution is interrupted for service by the monitor program. The interrupt sequence may be initiated by the object program itself (00006 instruction), by an I/O channel requesting service, or by an error conditic that requires monitor program service.

The monitor mode flag enables execution of the special instructions which are reserved for the monitor program. These are instructions 00030 through 00037. It also disconnects the Y and Z registers from their normal control paths and opens the entire storage range to the monitor program. Further I/O channel service requests are blocked until the monitor mode flag has been cleared.

The monitor mode flag is cleared by the monitor program execution of an 00006 instruction. This restores the Y and Z register functions and resume execution of the object program.

Floating Point Mode Flag -

The floating point mode flag is set to enable the interruption of an object program when a floating point computation goes out of range. The flag is set by execution of instruction 00003 and cleared by execution of instruction 00002. Detection of a floating point number out of range in a floating point functional unit sets the FP range overflow flag only if the floating point mode flag is set. This control is necessary so that an object program can process vector streams with out of range elements without the delay of unnecessary interruptions.

Interrupt Sequence

An interrupt sequence is the mechanism for terminating object program execution in order to service a monitor call. A monitor call may originate in the object program itself (00006 instruction), by an I/O channel requesting service, or by an error condition that requires monitor action. The mechanism for these calls is implemented in the flags in the M register. M register flags 55 through 61 may be set only in an object program mode (no monitor mode flag). When set these flags cause the initiation of the interrupt sequence. The interrupt sequence begins as soon as all program parcels have been released from the execute queues. No further instruction issue is allowed.

The interrupt sequence begins by setting the monitor mode flag. This prevents any further interrupt requests from being honored. The object program scalar accumulator content is stored in scalar register zero (see special role of scalar register zero). The next program address for the object program is stored in register 17 of the program exit stack. The monitor program entrance address is read from register zero of the program exit stack. Execution of the monitor program is initiated at this address. The monitor mode flag enables the execution of instructions 00030 through 00037 and disconnects the Y and Z registers from their normal control functions. This opens the entire storage range to the monitor program.

A monitor program terminates execution and returns to the interrupted object program by executing an 00006 instruction. This instruction, when executed in a monitor mode, clears the monitor mode flag, loads the scalar accumulator from scalar register zero, and reconnects the Y and Z registers for object program bounds control. The interrupted program is resumed by reading the content of register 17 in the program exit stack for the next program address. Any monitor service requests which occurred during the monitor program execution interval will now be honored and a second interrupt sequence will occur before execution of an object program instruction.

Monitor Call Flag -

The monitor call flag is set by object program execution of an 00006 instruction. The object program executes this instruction in order to call the monitor program for a service function. The scalar accumulator content is intended as the communication vehicle for details of the monitor call. The monitor program clears this flag after completion of the call and before returning to object program execution.

I/O Channel Request Flag -

The I/O channel request flag is set by an input or output channel control which requires monitor program service. This flag will only set during object program execution (no monitor mode flag). The requesting channel number may be determined by monitor program execution of the 00030 instruction. The monitor program clears this flag after completion of the call and before returning to object program execution.

FP Range Overflow Flag -

This flag is set by the detection of an out of range floating point number in a floating point functional unit. The setting of this flag is allowed only if the floating point mode flag is also set. The monitor program uses the presence of this flag to determine cause of interruption.

Program Bounds Overflow Flag -

This flag is set by the execution of an instruction fetch beyond the limit address set for the object program field (Z register). The instruction fetch is aborted in this case and this flag is set for monitor program determination of the cause of interruption.

Operand Bounds Overflow Flag -

This flag is set by the execution of an operand fetch or store instruction with an address which is beyond the limit address set for the object program field (Z register). Instruction execution is aborted in this case and this flag is set for monitor program determination of the cause of interruption.

Exit Stack Overflow Flag -

This flag is set by object program execution of a return jump instruction with the exit stack pointer, E , equal to 15. The return jump instruction is completed in a normal manner storing the return address in register 16. The object program is then interrupted and this flag is used by the monitor program in determining the cause of interruption.

Exit Stack Underflow Flag –

This flag is set by object program execution of an 00005 instruction with the exit stack pointer, E, equal zero. The entrance address for the monitor program is read from register zero of the exit stack and this flag is used by the monitor program to determine cause of interruption.

Object Program Zero Flag –

This flag is a storage place for a bit of the object program W register during the monitor program execution interval. This releases the W register for use by the monitor program.

Object Program Sign Flag –

This flag is a storage place for a bit of the object program W register during the monitor program execution interval. This releases the W register for use by the monitor program.

Special Role of Scalar Register Zero

Scalar register zero performs a special function in object program calls to the monitor program. The interrupt sequence stores the scalar accumulator content in this register in the process of initiating monitor program execution. This releases the scalar accumulator for use by the monitor program.

An object program reading scalar register zero as an operand register reads a zero value. This value is forced by the hardware to make the register useful in object programing. An operand stored in this register by an object program is discarded.

A monitor program reading scalar register zero as an operand register reads the value stored there by the interrupt sequence. This is the object program scalar accumulator content at time of interruption. This value has meaning to the monitor program if the cause of interruption was an object program monitor call. For other types of interruption of the object program the scalar register zero is simply a place to store the scalar accumulator content over the monitor program execution interval. A monitor program may store new data in scalar register zero. This is appropriate only for object program calls and is the responsibility of the monitor program software.

'Execute Next If' Instructions

00010 X, W = 0
00011 X, W ≠ 0
00012 X, W > 0
00013 X, W < 0

These four instructions allow program execution of the following parcel of instruction code if the designated condition is satisfied. All four conditions are based on the value of the W register. The first instruction allows execution of the following parcel if the W register contains a zero value. If not, the following parcel is treated as a pass. The second instruction listed above is the reverse test. The third instruction allows execution of the following parcel if the content of the W register is positive (sign bit zero). The last instruction is the reverse test for negative.

The parcel of program instruction code which follows the above instructions has special limitations. It must not be a branch instruction nor an instruction which alters the B or L register content. Specifically, it must be a parcel of code which can issue to an execute queue along with the 00010 through 00013 instruction. If the following parcel does not meet this criterion the 'execute next if' instruction becomes a pass.

A second restriction in the use of the above instructions relates to the position of the instruction in the 64 bit word. The 'execute next if' instruction must not be the last parcel of a word. If it is, it becomes a pass instruction as in the preceeding restriction. This second restriction results because of the need to interrupt at word boundaries.

These four instructions are included in the instruction list because they allow conditional execution as a normal part of the issue/execute mechanism. As such they are much faster than a conditional branch would be.

'Execute Next When' Instructions

164 X = VD
00164 X = V@

These two instructions allow execution of the next parcel of program code when the indicated vector register is free. All execution of code beyond this point in the program is delayed until the register free condition is satisfied. The first instruction listed above holds execution until the vector register designated by the D parameter in the instruction is free. The second instruction holds execution until the vector register designated by the B register content is free.

These instructions are included in the instruction list to interlock the storage references from the vector portion of the machine where required. Vector storage references may be out of sequence because there is no hardware test of each individual element address at instruction issue time. These instructions allow for insuring sequential execution where a possible conflict may occur.

CRAY-2 General Characteristics

The CRAY-2 computer characteristics differ from the CRAY-1 in several basic respects. The CRAY-1 program instructions use a three address format with very limited numbers of directly addressed operating register The CRAY-2 instructions use an accumulator for continuity between one address instructions which may reference a larger number of operating registers. This increases the total data bandwidth in the CRAY-2 by allowing more operations to proceed in parallel. Operations are identical in the functional units of the two machines and the differences appear in the use of the accumulator as one operand in CRAY-2 instructions.

A CRAY-2 instruction sequence generally begins with an instruction that loads the accumulator with an operand from an operating register. A second instruction then reads an operand from another operating register and performs a function with the previous accumulator content as the second operand. The result is left in the accumulator. A third instructic then stores the result in an operating register or continues the computat with another function. Simulation of a CRAY-1 program on a CRAY-2 machine requires three CRAY-2 instructions for the equivalent function of many CRAY-1 instructions. Compilation of a CRAY-1 machine language program for CRAY-2 execution can combine many functions and the result is more like a 50% increase in the total number of CRAY-2 instruction words over the CRAY-1 equivalent requirement. Speed in the CRAY-2 machine is accomplished by issuing more than one instruction per clock period and allowing more parallel execution of program sequences.

The scalar registers in the CRAY-1 are arranged in a hierarchy with the A and S registers directly addressed by the instructions and the B and T registers in a second level backup role. There are a total of 144 scalar registers in a CRAY-1 machine. The CRAY-2 scalar registers are merged in a common pool with address and operand registers all of 64 bit length. There are a total of 512 such registers which are addressed by a nine bit designator in the CRAY-2 instructions.

Storage References

A CRAY-1 machine has a single data path from the scalar and vector arithmetic area of the machine to the storage area. This path is greatly expanded in the CRAY-2 machine. The scalar portion of the CRAY-2 machine has its own data path to storage independent of the vector sectio The vector section has four read data paths and two write data paths which are independent of each other. It is then possible for four vector streams to be reading data from storage and two to be writing data at the same time. This necessarily means that there is no hardware interlock to prevent reading data from storage in a vector mode before data has been written there from a previously issued vector instruction. Such interlocking functions must be provided by software at compile time or by interlocking the vector execution at run time through a special instruction for this purpose.

Vector Registers

Vector registers in the CRAY-2 machine are similar to the vector registers of the CRAY-1. Each register has 64 elements of 64 bits each. Vector length in the CRAY-2 machine is specified in a hardware register similar to the CRAY-1 register. There are 32 vector registers in the CRAY-2 as vs eight in the CRAY-1. These vector operating registers are referenced by one address instructions similar to the scalar instructions. A vector program sequence utilizes a vector accumulator for continuity in a manner similar to the scalar sequence just described. The vector accumulator may be chained repeatedly in a single vector sequence and is therefore transparent in a simulated three address vector operation of the CRAY-1 type. The accumulator reduces the requirement for intermediate registers which limits the CRAY-1 vector bandwidth. The result is that the 32 vector registers of the CRAY-2 have a potential of more than four times the CRAY- bandwidth.

Exchange Package

The CRAY-1 machine utilizes an exchange package of 16 storage locations to store and restore the principle operating registers on a call to the monitor program. No such package exists in a CRAY-2 machine. An interrupt of a CRAY-2 object program for a monitor function stores only the scalar accumulator content as a part of the interrupt sequence. This makes the monitor call somewhat faster but requires that the monitor program must store and restore those operating register used.

Indirect Registers

The CRAY-2 machine has an instruction subset which uses a operating register as a register pointer to replace the direct register designator of the normal instruction set. These indirect register references allow program loops to scan the operating registers and thus save program storag

Program Addressing

The program code in the CRAY-2 is addressed by complete 64 bit words only. The parcel addressing of the CRAY-1 has the advantage of more compact code and shorter interrupt lockout time. These advantages are outweighed in the CRAY-2 by the effectiveness of the relative branching with the full word addressing. Program code in the CRAY-2 has no constant or absolute addresses in the body of the code other than the nine bit displacement designator. Constants and addresses are stored in a table associated with each subroutine. There is an advantage in having these addresses all of full word character rather than parcel addresses for instructions and word addresses for data.

All program entrance points must be left justified in the 64 bit word. Relative branching specifies the number of 64 bit words forward or backward from the word which contains the current instruction parcel.

Instruction Subsets A & B

Instruction subset A contains those instructions which do not involve a register designation. Instruction subset B is a variation on the primary instruction set for the last half of the list. A subset B instruction 001XX0 is equivalent to instruction 1XX000 with the role of the D designator replaced with the B designator. In each of these subsets the lowest order octal digit is reserved for control of the indexing registers which are used in small program loops. The bit positions in the lowest order octal digit are assigned the following roles.

Bit 13 - Advance A flag

Bit 14 - Advance B flag

Bit 15 - Reduce L flag

The indicated register content is modified after the instruction has been interpreted for execution. For those instructions which enter one of the control registers with data this indexing function may conflict. In such conflict cases the primary instruction function is implemented and the indexing function is ineffective.

Special Designators

The following designators are used for the indicated function in the instruction descriptions.

A = scalar accumulator content (64 bits)
AD - scalar register D content (64 bits)
Aj - scalar register B content (64 bits)
B - register pointer (9 bits)
Cj - channel B control
D - lower portion of instruction (9 bits)
E - exit stack pointer (4 bits)
F - upper portion of instruction (7 bits)
Gj - channel B current address (24 bits)
Hj - channel B limit address (24 bits)
I - interrupt channel data (6 bits)
J - floating point mode flag (1 bit)
L - function length parameter (9 bits)
M - program mode register (15 bits)
P - current program address (24 bits)
Q - monitor program call
R - subroutine call
T - clock period counter (64 bits)
V - vector accumulator content (64 x 64 bits)
VD - vector register D content (64 x 64 bits)
Vj - vector register B content (64 x 64 bits)
W - branch test register (2 bits)
X - execute next parcel conditionally
Y - program base address (24 bits)
Z - program limit address (24 bits)

Instruction Summary

000	Subset A	040	B = AD
001	Subset B	041	L = AD
002		042	B, L = AD
003		043	
004	B = D	044	AD = B
005	L = D	045	AD = L
006		046	AD = B, L
007		047	AD = P
010	R = AD	050	V@ = (A), AD
011	P = AD	051	V@ = (VD)
012	P = AD, L = 0	052	VD = (A), A@
013	P = AD, L ≠ 0	053	VD = (V@)
014	P = AD, W = 0	054	(A), AD = V@
015	P = AD, W ≠ 0	055	(VD) = V@
016	P = AD, W > 0	056	(A), A@ = VD
017	P = AD, W < 0	057	(V@) = VD
020	R = P + D	060	
021	P = P + D	061	
022	P = P + D, L = 0	062	
023	P = P + D, L ≠ 0	063	
024	P = P + D, W = 0	064	
025	P = P + D, W ≠ 0	065	
026	P = P + D, W > 0	066	
027	P = P + D, W < 0	067	
030	R = P - D	070	
031	P = P - D	071	
032	P = P - D, L = 0	072	
033	P = P - D, L ≠ 0	073	
034	P = P - D, W = 0	074	
035	P = P - D, W ≠ 0	075	
036	P = P - D, W > 0	076	
037	P = P - D, W < 0	077	

Instruction Summary

100	A = AD	140	V = VD
101	A = A ∗ AD	141	V = V ∗ VD
102	A = A + AD	142	V = V + VD
103	A = A - AD	143	V = V - VD
104	AD = A	144	VD = V
105	AD = (A)	145	VD = (A)
106		146	
107	(A) = AD	147	(A) = VD
110	A = D	150	M: AD = V(0)
111	A = A ∗ D	151	M: AD = V(F)
112	A = A + D	152	M: AD = V(+)
113	A = A - D	153	M: AD = V(-)
114	A = A > D	154	F: AD = V
115	A = A < D	155	
116	AD = V(A)	156	M: V = VD ∗ A
117	V(A) = AD	157	M: V = V + VD ∗ A
120	A = AD, A > L	160	V = VD, V > A
121	A = A, AD < L	161	V = V, VD < A
122	P: A = AD	162	P: V = VD
123	Z: A = AD	163	Z: V = VD
124	L: A = A ∗ D	164	X = VD
125	L: A = A ∗ AD	165	L: V = V ∗ VD
126	L: A = A + AD	166	L: V = V + VD
127	L: A = A - AD	167	L: V = V - VD
130	F: A = A + AD	170	F: V = V + VD
131	F: A = A - AD	171	F: V = V - VD
132	R: A = AD	172	R: V = VD
133		173	
134	F: A = A ∗ AD	174	F: V = V ∗ VD
135	G: A = A ∗ AD	175	G: V = V ∗ VD
136	H: A = A ∗ AD	176	H: V = V ∗ VD
137	I: A = A ∗ AD	177	I: V = V ∗ VD

Instruction Subset A

00000	Pass
00001	
00002	J = 0
00003	J = 1
00004	W = A
00005	P = (E)
00006	Q = A
00007	
00010	X, W = 0
00011	X, W ≠ 0
00012	X, W > 0
00013	X, W < 0
00014	E = A
00015	B, L = I
00016	B = A
00017	L = A
00020	A = E
00021	A = (E)
00022	A = C@
00023	A = L
00024	A = T
00025	A = M
00026	A = Y
00027	A = Z
00030	(E) = A
00031	C@ = 0
00032	C@ = A
00033	H@ = A
00034	T = A
00035	M = A
00036	Y = A
00037	Z = A
00040	V = A
00041	V = V * A
00042	V = V + A
00043	V = V − A
00044	
00045	
00046	
00047	
00050	
00051	
00052	
00053	
00054	
00055	
00056	M: V = V * A
00057	
00060	V = V > A
00061	V = V < A
00062	V = V >> A
00063	V = V << A
00064	
00065	L: V = V * A
00066	L: V = V + A
00067	L: V = V − A
00070	F: V = V + A
00071	F: V = V − A
00072	
00073	
00074	F: V = V * A
00075	C: V = V * A
00076	H: V = V * A
00077	

Instruction Subset B

```
00100   A = A@
00101   A = A * A@
00102   A = A + A@
00103   A = A - A@

00104   A@ = A
00105   A@ = (A)
00106
00107   (A) = A@

00110   A = B
00111   A = A * B
00112   A = A + B
00113   A = A - B

00114   A = A > B
00115   A = A < B
00116   A@ = V(A)
00117   V(A) = A@

00120   A = A@, A > L
00121   A = A, A@ < L
00122   P: A = A@
00123   Z: A = A@

00124   L: A = A * B
00125   L: A = A * A@
00126   L: A = A + A@
00127   L: A = A - A@

00130   F: A = A + A@
00131   F: A = A - A@
00132   R: A = A@
00133

00134   F: A = A * A@
00135   G: A = A * A@
00136   H: A = A * A@
00137   I: A = A * A@

00140   V = V@
00141   V = V * V@
00142   V = V + V@
00143   V = V - V@

00144   V@ = V
00145   V@ = (A)
00146
00147   (A) = V@

00150   M: A@ = V(0)
00151   M: A@ = V(#)
00152   M: A@ = V(+)
00153   M: A@ = V(-)

00154   F: A@ = V
00155
00156   M: V = V@ * A
00157   M: V = V + V@ * A

00160   V = V@, V > A
00161   V = V, V@ < A
00162   P: V = V@
00163   Z: V = V@

00164   X = V@
00165   L: V = V * V@
00166   L: V = V + V@
00167   L: V = V - V@

00170   F: V = V + V@.
00171   F: V = V - V@
00172   R: V = V@
00173

00174   F: V = V * V@
00175   G: V = V * V@
00176   H: V = V * V@
00177   I: V = V * V@
```