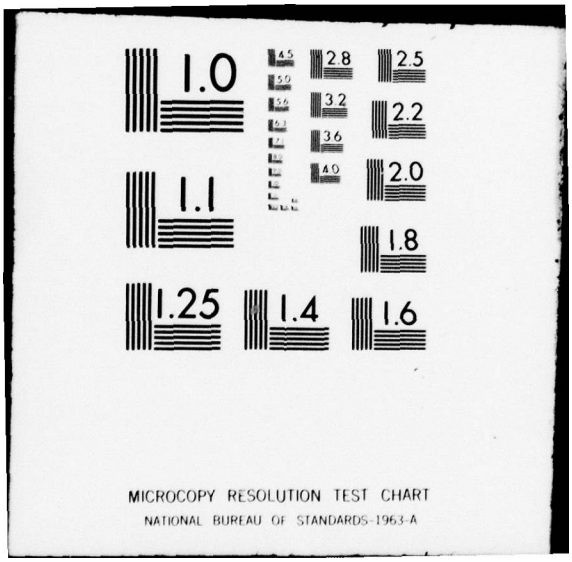
AD-A060 638 MICHIGAN UNIV ANN ARBOR SYSTEMS ENGINEERING LAB F/G 9/2
A CRAY-1 SIMULATOR.(U)
SEP 78 D A ORBITS
UNCLASSIFIED SEL-118 AFOSR-75-2812
AFOSR-TR-78-1405 NL

1 OF 2
AD
A060638




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A060638

DDC FILE COPY

78-1405

SEL Report No. 118

LEVEL II

(3)

A CRAY-1 Simulator

D. A. Orbits



September 1, 1978

Sponsored Jointly by
Directorate of Mathematical and Information Sciences,
Air Force Office of Scientific Research, and
Air Force Flight Dynamics Laboratory,
Wright-Patterson Air Force Base,
under Grant 75-2812



DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
SYSTEMS ENGINEERING LABORATORY
THE UNIVERSITY OF MICHIGAN, ANN ARBOR

78 10 16 122

Approved for public release;
distribution unlimited.

SECURITY CLASSIFICATION OF THIS PAGE (When Data

REPORT DOCUMENTATION P

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFOSR-TR-78-1405	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): A CRAY-1 SIMULATOR, SEL REPORT #118	5. TYPE OF REPORT & PERIOD COVERED: Interim rept.
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s): D.A. Orbits	8. CONTRACT OR GRANT NUMBER(s): AFOSR-75-2812
9. PERFORMING ORGANIZATION NAME AND ADDRESS: University of Michigan, Dept. of Electrical and Computer Eng., Ann Arbor, Michigan	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F 2304/A3 A3
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research/NM, Bolling AFB, Washington, DC 20332	12. REPORT DATE: 1 September 1978
	13. NUMBER OF PAGES: 125
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): SEL-118	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

123p.

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Vector Processing
Parallel Processing

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A Cray-1 logic-timing simulator is described. This simulator is written in Fortran-IV on an IBM 360/370 series machine. The simulator provides extensive reporting of Cray-1 resource usage and resource conflicts. By calling the simulator as a subroutine, the user may flexibly use Cray-1 program simulation within a larger problem environment. Extensive interactive debugging features make the Cray-1 simulator a powerful tool for the development of Cray-1 assembly language programs.

DD FORM 1 JAN 73 1473

400 704

UNCLASSIFIED

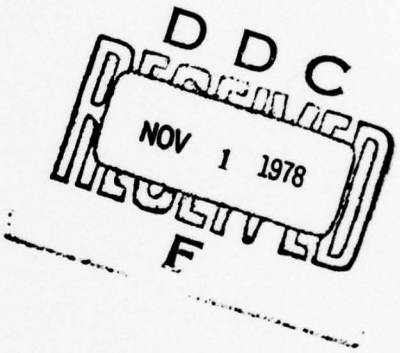
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(3)

A Cray-1 Simulator

D. A. Orbits

Systems Engineering Laboratory
University of Michigan
Ann Arbor, Michigan 48109
September 1, 1978



SEL Report #118

Sponsored Jointly by the Directorate of Mathematical and Information Sciences, Air Force Office of Scientific Research, and the Air Force Flight Dynamics Laboratory, under Grant 75-2812

This document has been approved for public release and sale; its distribution is unlimited.

78 10 16 122

Abstract

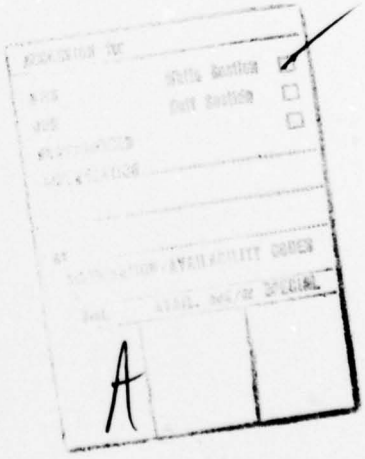
A Cray-1 logic-timing simulator is described. This simulator is written in Fortran-IV on an IBM 360/370 series machine. The simulator provides extensive reporting of Cray-1 resource usage and resource conflicts. By calling the simulator as a subroutine, the user may flexibly use Cray-1 program simulation within a larger problem environment. Extensive interactive debugging features make the Cray-1 simulator a powerful tool for the development of Cray-1 assembly language programs.

Acknowledgment

I gratefully acknowledge the aid and assistance of the following people.

- Professor D. A. Calahan, The University of Michigan
- William Ames, The University of Michigan
- Jerry Brost, Cray Research Inc.
- George Granander, Cray Research Inc.

Also, acknowledgement is given the National Center for Atmospheric Research for access to a CRAY-1 for confirmation of instruction timing.



Preface

The simulator described in this report was developed to support general vector algorithm studies and to prepare and evaluate performance of 3-dimensional aerodynamic fluid flow codes on a vector processor. It was felt to be of sufficient general interest and utility that this documentation was prepared.

The simulator will accept machine code from either of two cross assemblers: (1) a CDC 6000/7000 series version developed at the National Center for Atmospheric Research, and (2) an IBM 360/370 series version developed at the University of Michigan and described in companion SEL Report #120.

Table of Contents

Preface

1. Introduction 1
   - 1.1 Forward 1
   - 1.2 Summary 2
2. Simulator Features 4
   - 2.1 Command Language 5
   - 2.2 Exceptional Conditions 12
   - 2.3 Subroutine Interface 16
   - 2.4 CRAYEX Exit Dispatcher 20
   - 2.5 Report Generation 23
   - 2.6 Inconsistencies with the Cray-1 48
3. Examples 49
   - 3.1 Stand-alone use 49
   - 3.2 Subroutine use 51
4. Simulator Command Descriptions 53
5. Simulation Costs 89
6. Bibliography 92

Appendices

A. Summary of Cray-1 Timing Information 93
B. Simulator I/O Device Usage 98
C. Simulator Common Block Usage 99
D. Establishing the Simulator on MTS 100
E. Summary of Cray-1 Instruction Set 101
F. Simulator Error Messages and Error Stops 106
G. Program Availability Information 108
H. Sample Simulator Exit Dispatcher 109
I. Sample Simulator Calling Program 111
J. Glossary of MTS terms 115
K. Load Module Formats 117

1. Introduction

1.1 Forward

The University of Michigan Cray-1 simulator was written during 1977-78. The decision to build a simulator was motivated by the following considerations:

(1) Access to a Cray-1 for the purposes of algorithm design and code development was often very difficult and access on any continuing basis for research purposes was not possible. For most research purposes simulation will provide satisfactory performance.

(2) Even with access to a Cray-1, it is often quite difficult to analyze algorithm performance. This is because there is no hardware instrumentation on the Cray-1 to permit a study of CPU resource usage and conflict. The U of M Cray-1 simulator can provide a detailed report of CPU activity.

(3) With simulation it is possible to study the impact of Cray-1 architectural modifications on algorithm performance.

(4) For algorithms which must be carefully designed and coded, the programmer can use the simulator to analyze instruction delays and re-order instructions as necessary to minimize conflicts.

(5) When debugging programs, it is useful to have interactive control of program execution. Through the use of break-points, at-points and command files, the simulator lends considerable flexibility to the debugging process.

Experience with the Cray-1 simulator has shown it to be a very useful and flexible algorithm design and code development tool. The simulator was designed for use on the Michigan Terminal System (MTS). MTS is an interactive time-sharing system developed at the U of M and is presently running on an Amdahl 470/V6 computer. This machine is compatible with an IBM 370/168. As a result, the U of M Cray-1 simulator is mildly MTS dependent and heavily IBM 370 dependent. The MTS dependencies are related to I/O usage and the 370 dependencies are related to byte addressing and the 32 bit word of the 370 architecture. See appendix G. for program availability information.

1.2 Summary

We have used the Cray-1 simulator in some substantial code development work that has shaken out a number of problems. However, some incompatibilities with the Cray-1 hardware do exist and are noted in section 2.6.

The timing accuracy of the simulator is fairly good. We have observed a timing error on the order of ± 1/2% with programs that have been run on the Cray-1. We have also timed sixty six short instruction test segments, all of which time correctly. Without convenient access to a Cray-1, we have been unable to further study the timing error.

The Cray-1 simulator currently provides two kinds of reporting:

(1) A summary report that includes vector unit busy times, floating point operation counts, average vector length calculation and data flow accounting.

(2) A resource activity report which permits a study of resource conflicts at each clock period of simulation.

For more information on reporting, see section 2.5.

The Cray-1 simulator can be flexibly integrated within a large problem environment through the following two features:

(1) The simulator may be called as subroutine. This permits the user to embed simulation within a larger program, allowing kernels or special functions to be simulated while leaving the rest of the program in Fortran.

(2) The simulator may call user defined subroutines to perform special functions (eg. I/O, SQRT etc.) that may be already available on the Cray-1.

See section 3 for an example of this.

As a front-end to the Cray-1 simulator, the command language provides the user interactive control over the simulation. The user may set break points and display or alter the registers and memory of the simulated Cray-1. As implemented at the University of Michigan, the simulator detects floating point operation exceptions, operand reference errors and branch errors and returns control to the user for investigation.

The Cray-1 simulator also provides a means for studying architectural modifications to the Cray-1. One option currently implemented is the ability to specify an improved memory bandwidth. With this option enabled, vector loads and stores move data to and from main memory at the fastest data rate possible consistent with the address increment. Depending on the increment, the data rate ranges between four words per clock period and 1/4 word per clock period. As algorithm study on the Cray-1 develops, the simulator can be used to assess the impact of Cray-1 architectural changes on algorithm performance.

2. Simulator Features

The U of M Cray-1 simulator was designed to time, as accurately as possible, the instruction resource usage of the Cray-1 vector processor. This section of the user manual has been divided into six sub-sections, each devoted to a particular aspect of the Cray-1 simulator. No attempt has been made to describe the architecture of the Cray-1 itself. The bibliography lists several sources for this information.

The following is an overview of the material covered in this section:

(1) Sub-section 2.1 includes an introduction to the simulator command language and the running of simulated programs.

(2) Sub-section 2.2 covers the exceptional conditions that may arise when using the simulator. For example, keyboard attention interrupts, arithmetic overflows, invalid simulator memory references, etc.

(3) Sub-section 2.3 covers the subroutine interface through which a Fortran program may call the Cray-1 simulator. This is useful for simulating only a portion of a program, while retaining the rest of it in Fortran-IV for either cost or convenience reasons.

(4) Sub-section 2.4 covers the simulator exit processing. Through the Cray-1 Exit instruction the user may have the simulated program call a user provided subroutine to perform functions that might be provided by the operating system or the subroutine libraries in an actual Cray-1 environment.

(5) Sub-section 2.5 covers the report generation facilities of the simulator. This reporting is controlled by the CPACT and STAT commands.

(6) Sub-section 2.6 covers inconsistencies between the simulator and the Cray-1 computer that are presently known. Unimplemented instructions are discussed here along with other minor inconsistencies such as data formats, timing inaccuracies, etc.

2.1 Command Language

The command language provides the user interface to the Cray-1 simulator. Through the command language, the user controls and monitors the progress of the simulated program. The user has considerable flexibility in controlling input to and output from the simulator. This section is organized into the following four subsections:

(1) Command language input control
(2) Command language output control
(3) Simulator control
(4) Running programs on the Cray-1 simulator

2.1.1 Command Language Input Control

When starting up the simulator in stand-alone mode (by the MTS command - $RUN CRAY1), the simulator will prompt for terminal input by typing a period. The user may then enter a command or redirect the command input stream to read from a file via the USE command. The filename parameter on the USE command directs the simulator to open that file and begin reading commands. Upon an end-of-file condition the input stream is switched back to the terminal.

More than one USE command may be issued, allowing nested command files to be built by the user. The simulator command language maintains a command stream input stack which controls the issue of nested USE commands.

The command stack is also used when the simulator is called as a subroutine (see section 2.3). For subroutine usage, the caller supplied command string is split at the command separator character (a semi-colon) and each command is written to a scratch file. This scratch file is termed the call-file. The call-file is terminated with a RETURN command, so that after execution of the caller commands automatic return is made from the simulator to the caller. After creating the call-file, the subroutine interface pushes the call-file onto the command stack causing subsequent commands to come from the call-file.

Another use of the command stack arises from the use of AT points that may be set by the user. An AT point is similar to a break point, in that each is set at some instruction address in the user's program. Upon hitting a break point, program simulation is halted and control reverts to the terminal allowing the user to monitor the program's behavior. An AT point differs, in that when it is created the user may also enter one or more simulator commands that will be automatically executed when the AT point is hit. These commands are saved in a scratch file and then, during simulation when the AT point is hit, the simulator pushes the AT point's scratch file onto the command stack causing subsequent commands to come from the AT file. A RUN command is automatically placed at the end of the AT file, causing simulation to resume uninterrupted after the AT commands have been processed. AT commands are useful for automatically displaying register or memory locations at selected points in a program. In cases where the user wishes to display various locations and then regain control for other purposes, entering the command USE *MSOURCE* will switch command input to the terminal during AT command processing. Any end-of-file condition ($ENDFILE or control C) will terminate input from the top entry of the command stack, causing the stack to be popped and input to continue from the previous source. In the case of an AT file with a 'USE *MSOURCE*' command in it, an end-of-file condition from the terminal will resume simulation. In fact, when a break point is hit, the simulator automatically issues an implied 'USE *MSOURCE*' command which reverts control to the terminal.

The command stack is fifteen levels deep with the base entry preset to *MSOURCE* which can never be popped. Only one AT or BREAK point can be hit at any time, therefore a subsequent RUN command will pop the command stack through the last AT or BREAK entry on the stack. Upon a RETURN command the command stack will be popped through the last call-file entry on the stack.

Occasionally due to an error condition the message "Command Stack Reset" will be printed. This means that the command stack has been cleared to the base entry which is preset to *MSOURCE*. This assures that the error condition will return input control to the user.

However, this means that any commands not yet executed in any outstanding call-files, AT files or USE files have been lost.

A keyboard attention interrupt will cause the command stack to be reset. This is useful to stop a USE file or prevent subsequent commands in the call-file from being processed.

2.1.2 Command Language Output Control

Normal output from the simulator (informational messages, DISPLAY output, etc.) can be sent to another file or device by using the SET command to switch the output device. For example, SET OUTPUT = *PRINT*, would route the output to the printer.

Error messages are output on a different unit number and always go to *MSINK* (the terminal). If an error situation arises causing the message "Command Stack Reset" to appear, the output device will be switched back to *MSINK*, if it was diverted elsewhere. Also, a keyboard attention will switch the output back to *MSINK*.

2.1.3 Simulator Control

To keep the simulator from running away from the user, a keyboard attention interrupt can be signalled which has the following effects:

(1) Resets the command input stack to read from *MSOURCE* (the terminal), losing any outstanding command files.

(2) Resets the output device back to *MSINK* (the terminal)

(3) Performs the following command dependent actions:

3.1) For a DISPLAY command, an attention will terminate the output. This is useful if a long display region was accidently displayed.

3.2) For a HELP or STAT command, an attention will terminate the output.

3.3) For a RUN command, an attention will stop the simulation and print the parcel address of the next instruction to be executed. Simulation may be resumed without any loss of timing information by just entering a "RUN" command. No parcel address should be supplied on the RUN command, as this always forces a buffer fetch which will make the timing inaccurate.

4) If for any reason the simulator seems to be looping and not responding to attentions, two attentions will return control to MTS.

Attention trapping is only enabled while control is inside the simulator or the command language. That is, if the simulator is called as a subroutine, attention trapping is enabled only while a call to the CRAY1 interface subroutine is active.

2.1.4 Running Programs on the Cray-1 Simulator

Before a program may be run on the simulator, it must first be translated to a format acceptable for loading into the simulator. This translation is typically done via a Cray-1 cross assembler. This assembler generates absolute or relocatable load modules that can be loaded by the simulator LOAD command. The format of the load module is described in appendix K.

When designing a Cray-1 program to be simulated, consideration must be given first to the nature of the algorithm under study. If the program requires some initialization which won't be written in Cray-1 assembly language, then perhaps the simulator should be called as a subroutine. It is possible for the calling program and the simulator to both share the Fortran common block that is used for the simulated Cray-1 memory. In fact, if the users calling program is loaded first by MTS, the user may increase the size of the simulated Cray-1 memory beyond the 4096 IBM double words that are presently allocated. For example, if the calling program was in the MTS file MAIN.O, the MTS command to start up the program would be

```
$RUN  MAIN.O+CRAY1
```

The file CRAY1 contains the object module of the simulator. See section 2.3 for a discussion about calling the simulator as a subroutine. In this example, execution would begin in the user's main program which would initialize the data area in the Cray-1 memory as required by the algorithm to be simulated. The main program would then call the subroutine CRAY1, passing a simulator command string which could load the Cray-1 program and begin simulation.

If the algorithm under study requires the use of intrinsic functions, such as SQRT, SIN, COS, etc, which would be supplied by some Cray-1 subroutine library, the user may provide these functions through the use of Cray-1 Simulator EXIT instruction dispatcher. The Cray-1 EXIT instruction (assembler mnemonic EX exp) contains a 9 bit expression field. If this field is non-zero the simulator will call a subroutine called CRAYEX, passing the value of the expression field and several register arguments to it. The user may write a CRAYEX subroutine to process these EXIT codes and perform any function he wishes to define. For example, an EXIT code of one could be defined to perform a square root operation. This EXIT feature avoids the expense of simulating Cray-1 code for such intrinsic functions by allowing them to be programmed directly on the host machine. See section 2.4 for a complete discussion of the EXIT dispatcher.

Several other differences between the Cray-1 computer and the simulator arise due to the nature of the IBM 370 architecture upon which the simulator runs.

To speed the simulation of arithmetic, all the arithmetic is done using the IBM 370 arithmetic instructions. The alternative would be to simulate the Cray-1 arithmetic, further raising the simulation cost. As a consequence of using host machine (IBM 370) arithmetic, integers in the simulator are 32 bits wide whereas Cray-1 integers are only 24 bits wide. Although the simulator memory word is a 64 bit 370 double word, which is the same width as a Cray-1 memory word, the floating point data format is different. On the Cray-1 the sign and exponent field is 16 bits wide whereas on the IBM 370 it is only 8 bits wide. Further, the Cray-1 exponent is a base 2 exponent whereas the IBM 370 exponent is base 16. Figure 2.1.1 shows the different formats.

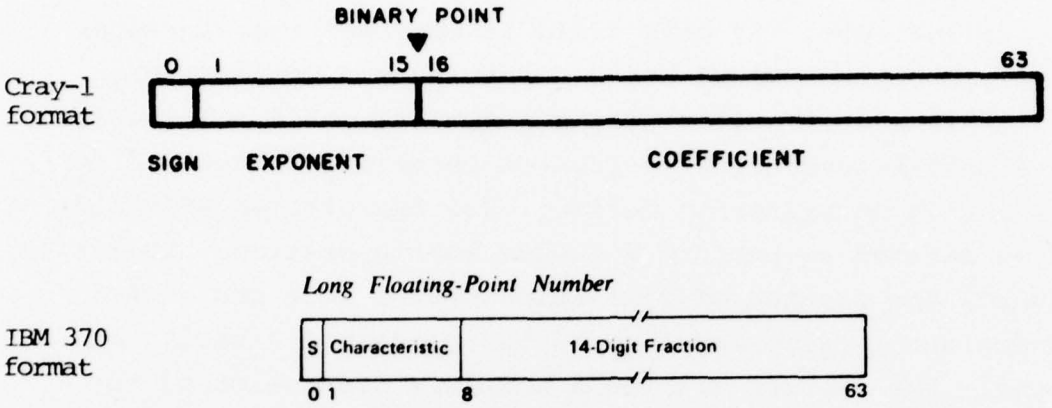


Figure 2.1.1 - Cray-1 vs. IBM 370 Floating point data formats

The simulation of the instruction computation is done in it's entirety when the instruction issues. The pipeline data flow in the Cray-1 is not simulated. This means that upon hitting a BREAK or AT point, all results of prior instructions are available for inspection or modification. The instruction where the BREAK or AT point is set has not yet been executed.

There are three methods for controlling the simulation of a Cray-1 program:

(1) BREAK points

(2) AT points

(3) An instruction issue limit parameter.

BREAK and AT points may be set at a specified parcel address in the simulated program. Setting BREAK or AT points do not change the instruction at that location, rather, BREAK and AT points are detected by monitoring the P address register. This permits BREAK and AT points to be set before the program is loaded or reloaded.

When a BREAK point is hit, control goes to the terminal. When an AT point is hit, a predefined command file is processed which was created when the AT point was set. Control will not go to the terminal when an AT point is hit if no command causes this to happen.

An instruction issue limit may be provided as an optional parameter on the simulator RUN command. For example, the following RUN command would begin execution at parcel address 21A and cause control to return to the command language after 2500 Cray-1 instructions have been issued (unless an EXIT instruction or error condition occurred).

```
RUN  21A  #2500
```

The issue limit parameter is a decimal number prefixed by a pound sign. If no issue limit is specified the remaining amount of a previous limit is used (in the case of a BREAK or AT or attention). If there is no remaining amount, a default value of 1000 is used. To single step through a program use the command:

```
RUN #1
```

Without a parcel address specified, simulation proceeds with the instruction pointed at by the P address register.

While in the command language, the user may display or change registers and memory locations by using the DISPLAY and CHANGE commands.

The HELP command may be used if you are unsure of command syntax or the function of a command. For example

```
HELP  RUN
```

will describe the simulator RUN command.

See section 4 for command descriptions of all simulator commands.

The cost of simulating Cray-1 programs is an important factor. The simulator provides three levels of cost control:

- Level 1 - Result computation only, still allows debugging but eliminates the cost associated with timing the Cray-1 instructions.
- Level 2 - Timing enabled, allows the timing of the simulated program at a cost of about 10 times the level one cost.
- Level 3 - CPACT (clock period activity report) enabled, increases the cost to about 30 times the level one cost.

Section five treats the cost issue further.

2.2 Exceptional Conditions

While simulating a Cray-1 program, the simulator may encounter any of several exceptional conditions which will halt the simulation. The six possible exceptional conditions are listed below followed by a discussion of each one:

1) Error exit
2) Program range error
3) Operand range error
4) Floating point interrupt
5) Invalid instruction executed
6) Attention interrupt.

An occurance of any exceptional condition will reset the command stack and switch OUTPUT back to the terminal if it was diverted elsewhere. The name of the routine being executed will be displayed, if possible.

2.2.1 Error Exit

An error exit is caused when the Cray-1 executes a zero op-code. The simulator signals this condition by printing the message:

ERROR EXIT AT - p-addr

where p-addr is the parcel address of the error exit instruction. Since memory is initialized with zeros when the simulator is started up, a bad or missing branch could cause an error exit.

2.2.2 Program range error

A program range error is caused by a branch instruction which attempts to jump outside the limits of the currently defined simulator memory. If used stand-alone, 4096 words of simulator memory are available. A program range error is signalled by the message,

PROGRAM RANGE ERROR.
BRANCH AT bch-p-addr
TARGET ADDRESS WAS tar-p-addr
MEMORY SIZE IS msize

The parcel address of the offending branch instruction is given by the bch-p-addr field. The invalid target address of the offending branch instruction is given by the tar-p-addr field. The memory size (msize) is printed in octal for comparison with the invalid target address and to inform the user of the current memory size.

If the user has tried to extend the size of the simulated Cray-1 memory by loading a longer common block, he must inform the simulator of this by setting the MEMSIZ word in the MSIZE common block to the correct size of the Cray-1 memory (see section 2.3). If the user forgets to do this a size of 4096 is assumed which may cause the program range error.

2.2.3 Operand range error

An operand range error is caused by an operand load or store that exceeds the limits of the currently defined simulator memory. If used stand-alone, 4096 words of simulator memory are available. An operand range error is signalled by the message,

```
OPERAND RANGE ERROR AT P = p-addr
MEMORY SIZE IS msize
```

The parcel address of the offending memory reference instruction is given by the p-addr field. The memory size (msize) is printed in octal to inform the user of the current memory size. The comments above (under program range error), about user extension of Cray-1 memory, apply here as well.

A vector load or store to memory can cause an operand range error in several ways:

1) The base address may be out of range
2) The operand increment may be too large
3) The vector length may be too large.

2.2.4 Floating point interrupt

The floating point interrupt exception is handled differently by the simulator than it is on the Cray-1. This discussion will deal with the simulator response to a floating point interrupt. See the

Cray-1 Reference Manual for the Cray-1 response.

The simulator response to a floating point interrupt is a consequence of the behavior of the IBM 370 architecture. Three types of floating point interrupts may occur:

1) Exponent overflow
2) Exponent underflow
3) Division by zero.

All three types of floating point interrupt may be suppressed if the floating point interrupt bit in the Cray-1 mode register is clear. When the simulator starts up, this mode register bit is set, thereby enabling all three types of floating point interrupts. The setting of this mode register bit may be controlled by the user in two ways:

1) Through the SET EFI = [ON/OFF] command, the user may enable or disable floating point interrupts.
2) Through the Cray-1 instructions EFI and DFI, the program may enable or disable floating point interrupts.

Only one floating point interrupt is detected for each instruction simulated. This means that if a vector instruction causes 20 exponent overflows, only one will be detected. After the instruction has finished executing the simulator will announce the floating point exception (if the EFI mode bit is set) and return to the command language.

When an exponent overflow occurs, the following message is printed:

```
** EXPONENT OVERFLOW **
FLOATING POINT ERROR AT P = p-addr
```

When an exponent underflow occurs, the following message is printed:

```
** EXPONENT UNDERFLOW **
FLOATING POINT ERROR AT P = p-addr
```

When a division by zero occurs, the following message is printed:

** FLOATING POINT DIVIDE CHECK **
FLOATING POINT ERROR AT P = p-addr

For each of the three messages the parcel address (p-addr) of the instruction causing the interrupt is printed.

2.2.5 Invalid instruction executed

The monitor mode Cray-1 instructions are not implemented on the simulator. When one of these is executed, the simulator will print the message,

** ATTEMPT TO EXECUTE INVALID INSTRUCTION AT : p-addr

will be printed and the simulator will return to the command language. The offending instruction's parcel address (p-addr) is printed to aid in finding the instruction.

2.2.6 Attention interrupt

To stop the simulation or regain control during command file processing, the MTS terminal user may issue a keyboard attention by hitting the break key or a control-E. This attention interrupt will reset the command stack and halt simulation if in progress. If simulation was in progress the message,

** SIMULATOR ATTN AT P = p-addr **

will be printed, where p-addr is the instruction to execute next if simulation is continued. An attention will cause no information to be lost and simulation may be resumed, as if never interrupted, by entering a RUN command without a p-addr parameter.

2.3 Subroutine Interface

The Cray-1 simulator may be called as a subroutine from a user Fortran-IV program. Three benefits provided by this interface are:

1) Being able to convert only a portion of a Fortran program to Cray-1 assembly language, allows you to simulate the converted portion while leaving the remaining in Fortran to run more efficiently on the host machine.
2) Being able to enlarge the amount of simulated Cray-1 memory by extending the memory common block in the user's calling program and loading this program first. This avoids the need for recompiling the simulator.
3) When studying a given algorithm, for application to the Cray-1, it is convenient to perform any housekeeping and initialization functions in the user's Fortran program. Therefore, only the algorithm need be coded in Cray-1 assembly language.

This section will discuss the protocol used to communicate with the simulator from a calling program. This communication has two aspects to it: (1) the subroutine interface used to pass commands and control to the simulator and (2) the shared Cray-1 memory interface used to pass data to and from the simulator.

2.3.1 Simulator subroutine call

To access the simulator as a subroutine the following Fortran subroutine call is used:

```
CALL CRAY1('cmd[;cmd] ... !',echosw)
```

The first argument is a literal string enclosed by apostrophes which may be composed of one or more simulator commands. Each command follows the same syntax as the commands described in section 4. To specify multiple commands with a single call to the simulator, separate the commands with a semicolon. The entire command string must be terminated with an exclamation point and may not exceed 200 characters.

The second parameter (echosw) is a logical constant or variable. This parameter controls the echoing of the commands passed in the first argument. If echosw is .TRUE., the commands will be echoed to the current simulator output device as they are processed. If echosw is .FALSE., command echoing is suppressed.

If the user wants to give control to the terminal at some point in the command string sequence, the command USE *MSOURCE* will allow additional commands to be read from the terminal. For example, the call

```
CALL CRAY1('LOAD TRIDEC;USE *MSOURCE*;RUN 21A #2000!',.TRUE.)
```

will cause the file TRIDEC to be loaded into the simulator memory after which,the USE command will cause control to go to the user's terminal, allowing breakpoints to be set, etc. An end-of-file condition at the user's terminal (via ENDFILE, control-c, etc.) will terminate the USE command permitting the "RUN 21A #2000" command to be executed. When the last command in the command string is executed an automatic return is made to the caller of the simulator. By setting the echosw parameter to .TRUE., the three passed commands will be echoed to the simulator output device as they are processed.

In order to call the simulator as a subroutine, the user's program must first get control. To accomplish this, two things must be done:

1) The user program must be set up as a main program.
2) The user program must be loaded before the simulator is loaded.

This is a consequence of the following two facts:

1) When MTS starts up a Fortran program (via the MTS $RUN command), control is given to the main program.
2) When the MTS loader encounters more than one main program it ignores all but the first one.

The simulator has a small internal main program which gets control if the simulator is run stand-alone. But, if the user writes a main program and loads it before the simulator is loaded, the simulator's main program is ignored by the loader. Therefore, when loading is finished MTS will give control to the user's main program.

As an example, suppose the user wrote the following Fortran main program and compiled it into the MTS file MAIN.O.

```
CALL CRAY1('USE *SOURCE* !',.FALSE.)
STOP
END
```

To use this main program and have it get control first, use the following MTS run command:

```
$RUN MAIN.O+CRAY1
```

Although most user main programs would be more complicated than this one, this main program is in fact the small internal main program used by the simulator.

2.3.2 Simulator memory sharing

The simulated Cray-1 memory can be shared both by the simulator and the user's calling program. This is accomplished by having the user include in his program the Fortran common block declaration used by the simulator to allocate the Cray-1 memory space. This commom block declaration appears in the simulator as follows:

```
DOUBLE PRECISION MEM
COMMON /MEMORY/ MEM (4096)
COMMON /MSIZE/ MEMSIZ
INTEGER IMEM(2,1)
EQUIVALENCE (MEM(1), IMEM(1,1))
```

The MSIZE common block contains the single word MEMSIZ whose value is the current size of Cray-1 memory. MEMSIZ is used to perform bounds checking on branches and memory references made by the simulated Cray-1 instructions. When the simulator is called for the first time some once-only initialization is done which includes zeroing all of Cray-1 memory (MEM). Therefore, MEMSIZ must be initialized properly before the first call to CRAY1. Further since the once-only initialization will zero Cray-1 memory, the very first call to CRAY1 must be made before the user's calling program initializes any of MEM. It is suggested that this first initialization call be made as follows:

```
CALL CRAY1('INIT!',.FALSE.)
```

The MEMORY common block contains the array MEM, which is used as the Cray-1 memory by the simulator. This is declared in the simulator to be 4096 double words long. The user may extend this common block to enlarge the Cray-1 memory. This is done by writing a Fortran main program which includes the common declaration statements shown above, but with the 4096 constant replaced with a larger value as needed. Then by loading the user main program first (see section 2.3.1), the user's main program not only replaces the simulator's main program, but the user's enlarged version of the MEMORY common block replaces the simulator's version.

To pass data to and from the simulator Cray-1 memory, the user need only read and write data to the MEM array. However, because the Cray-1 memory address starts at location zero and Fortran arrays are indexed beginning at one, the user must formulate the index into MEM by using the Cray-1 memory address and adding one to it. For example, Cray-1 memory location 3 is MEM(4).

The following example is the skeletal structure of a user main program which extends Cray-1 memory to 8192 words.

```
      DOUBLE PRECISION MEM
      COMMON /MEMORY/ MEM(8192)
      COMMON /MSIZE/ MEMSIZ
      INTEGER IMEM(2,1)
      EQUIVALENCE(MEM(1),IMEM(1,1))
C
C .... SET UP MEMSIZ WITH THE NEW MEMORY SIZE.
      MEMSIZ = 8192
C
C .... DO SIMULATOR ONCE-ONLY INITIALIZATION
      CALL CRAY1('INIT!',.FALSE.)
            :
            :  User initialization of Cray-1 memory
            :
      CALL CRAY1('LOAD UPROG;RUN 20A #1000!',.TRUE.)
            :
            :  User prints out results of simulated computation
            :
      STOP
      END
```

See section 3.2 for a more complete example of accessing the simulator as a subroutine.

2.4 CRAYEX Exit Dispatcher

As discussed in section 2.3, it is often useful to allow the Cray-1 simulation to be embedded as a portion of a larger Fortran program. Conversely, it is also useful to be able to call a Fortran program from within the simulated Cray-1 program. This transfer of control from the Cray-1 program to a Fortran program is accomplished through the use of the Cray-1 exit instruction.

The Cray-1 assembly language mnemonic for the exit instruction is shown below:

```
EX    ijk
```

The exit code field (ijk) is a nine bit field within the exit instruction. Exit codes may range from zero to 511 decimal. When the simulator encounters an exit instruction, it checks the exit code field (ijk) for a non-zero value. If ijk is zero, a normal Cray-1 program exit is performed. If ijk is non-zero, the simulator will call the subroutine CRAYEX. If the user supplies a CRAYEX subroutine and loads it first (see section 2.3.1), the user's CRAYEX routine will get control. If no user CRAYEX routine is provided, the simulator will perform a normal Cray-1 program exit.

If the user provides a CRAYEX routine the simulator will call it with the following Fortran subroutine call statement:

```
CALL CRAYEX(IJK, AREG, SREG, VREG, VL, EXSW)
```

The arguments passed by the subroutine call are discussed below:

IJK - This input parameter is an integer which contains the value of the ijk field in the exit instruction. It may be used as a dispatch parameter, allowing different exit codes to perform different functions.

AREG - This parameter is an eight element integer array used to pass the Cray-1 A-register contents to CRAYEX. This allows arguments to be provided and results returned through the A-registers. Cray-1 register AØ corresponds to AREG(1).

SREG - This parameter is an eight element double precision array used to pass the Cray-1 S-register contents to CRAYEX. This allows arguments to be provided and results returned through the S-registers. Cray-1 register SØ corresponds to SREG(1).

VREG - This parameter is a double precision array, dimensioned as (64,8), used to pass the Cray-1 Vector register contents to CRAYEX. Arguments may be provided and results returned through the vector registers. Cray-1 vector register VØ corresponds to VREG(~,1).

VL - This parameter is an integer which contains the value of the Cray-1 vector length register. On entry to CRAYEX, VL will always be between one and 64. VL may be changed by CRAYEX and this change will be reflected in the Cray-1 vector length register. On return from CRAYEX to the simulator VL must be in the range of 1 to 64.

EXSW - This return parameter is of type logical. On return to the simulator from CRAYEX, if EXSW is .TRUE., the simulator will perform a normal Cray-1 program exit. If EXSW is .FALSE., the simulator will treat the exit instruction as a no-op and continue the simulation with the next instruction. On entry to CRAYEX, EXSW has the value .FALSE..

In addition to the CRAYEX calling parameters, the CRAYEX program may access Cray-1 memory by sharing the memory common block as described in section 2.3.2. This permits the CRAYEX routine to perform major computation, I/O etc., directly to the Cray-1 memory.

The following example is a skeleton CRAYEX dispatcher.

```
      SUBROUTINE CRAYEX(IJK, AREG, SREG, VREG, VL, EXSW)
      LOGICAL EXSW
      INTEGER AREG(8), VL
      DOUBLE PRECISION SREG(8), VREG(64,8)
C
C ... DISPATCH ON THE EXIT CODE.
      GO TO (100, 200, 300, ...), IJK
C
C ... EXIT CODE UNDEFINED - TREAT AS NORMAL EXIT
      EXSW = .TRUE.
      RETURN
C
C ... EXIT CODE = 1.
100   .
      .  do exit code 1 processing.
      .
      RETURN
C
C ... EXIT CODE = 2
200   .
      .  do exit code 2 processing.
      .
      RETURN
      .
      .
      .
      END
```

The user may define exit code one to be a SQRT function, exit code 2 to be COS function, etc. Arguments and results may be passed through the registers or memory providing considerable flexibility in the algorithm design and implementation.

2.5 Report Generation

The Cray-1 simulator produces two kinds of report outputs, STAT and CPACT. The STAT report is a summary report of the program's use of Cray-1 resources. The CPACT report is a detailed report of Cray-1 resource usage at each clock period of the program's execution.

2.5.1 STAT report

The STAT report consists of three sections:

1) Vector Usage Counts
2) Floating Point Result Counts
3) Data Traffic Counts

Timing must be enabled only for the Vector Usage Counts section.

The Vector Usage Counts section reports the program's use of the Cray-1 vector unit resources. Figure 2.5.1, below, shows the Vector Usage Counts table.

```
                    U OF M  CRAY-1  SIMULATOR   (UN138) THU JUN 15/78

                              VECTOR  USAGE  COUNTS

CUM. TIMING        FP ADD   FP MUL   FP DIV    LOG.    SHIFT   I. ADD   V-LOAD   V-STOR

TIME BUSY (CP)       5882     7705       67      67     1277        0    10322     2501
% TIME BUSY        36.20%   47.42%    0.41%   0.41%    7.86%    0.0 %   63.52%   15.39%
NO. RESULTS          5530     7245       63      63     1201        0     9706     2316
NO. VECTORS            88      115        1       1       19        0      154       37
AVERAGE VL          62.84    63.00    63.00   63.00    63.21      0.0    63.03    62.59

RUN TIME (CP) :  16250
MFLOPS        :  62.67
COMPOSITE AVL :  62.95
CONCURRENCY   :   1.71
MIPS          :   5.02
```

Figure 2.5.1 - Vector Usage Counts Table

Each column of the Table represents a different vector functional unit. Left to right the units are: floating point add, floating point multiply, floating point reciprocal approximation, logical, shift, integer add and memory, split between vector loads and vector stores. The rows of the table represent: unit busy time, percent unit busy of total run time, the number of results produced by the

unit, the number of vector instructions issued to the unit and the average vector length processed by the unit.

Five other statistics are printed beneath the table: the run time since the last INIT command or simulator start up, the MFLOPS (million floating point operations per second) for the program, the composite average vector length over all vector units, the vector unit concurrency, and the MIPS rate.

MFLOPS is calculated over all floating point operations, both vector and scalar. It is computed as the number of floating point operations divided by the program run time in seconds.

Concurrency is calculated as the sum of all vector unit busy times divided by the program run time. It is a global measure of the concurrent use of the Cray-1 vector units.

MIPS, millions of instructions per second, is calculated as, the number of instructions issued divided by the program run time in seconds.

The Floating Point Result Counts section reports the program's use of both vector and scalar floating point operations. For each entry in the table (Figure 2.5.2) both the number of results and it's percentage are printed.

FLOATING POINT RESULT COUNTS

	ADDITION	MULTIPLICATION	RECIPROCAL	TOTAL
VECTOR (%)	5530 (43.4)	7119 (55.9)	63 (0.5)	12712 (99.9)
SCALAR (%)	5 (0.0)	5 (0.0)	8 (0.1)	18 (0.1)
TOTAL (%)	5535 (43.5)	7124 (56.0)	71 (0.6)	12730 (100.0)

Figure 2.5.2 - Floating Point Result Counts Table

Floating point additions (and subtractions) and reciprocals are counted directly from the instructions that perform them, but the multiplication count requires some adjustment due to the reciprocal approximation.

Because a reciprocation on the Cray-1 is an approximation, two additional multiplications must be done to get a full precision result. One of these multiplications is a reciprocal iteration and the other is a standard multiplication. The Cray-1 instruction sequence below illustrates the scalar instructions used to obtain a full precision scalar reciprocal (S1 = 1/S2).

```
S1  /HS2      reciprocal approximation
S2  S2*IS1    reciprocal iteration
S1  S2*S1     extend precision
```

To count these additional multiplies as part of the floating point operation count would overstate this count, since they really are part of a single reciprocal operation. Consequently these two multiplies have been deducted from the multiply count in the table. This adjustment is made by subtracting the number of detected reciprocal iterations from the number of standard multiplications. A reciprocal iteration is detected through the issue of a 067, 166 or 167 instruction. The sum of all vector and scalar floating point operations, shown in the lower right corner of the Figure 2.5.2, is used as the numerator of the MFLOPS calculation discussed above.

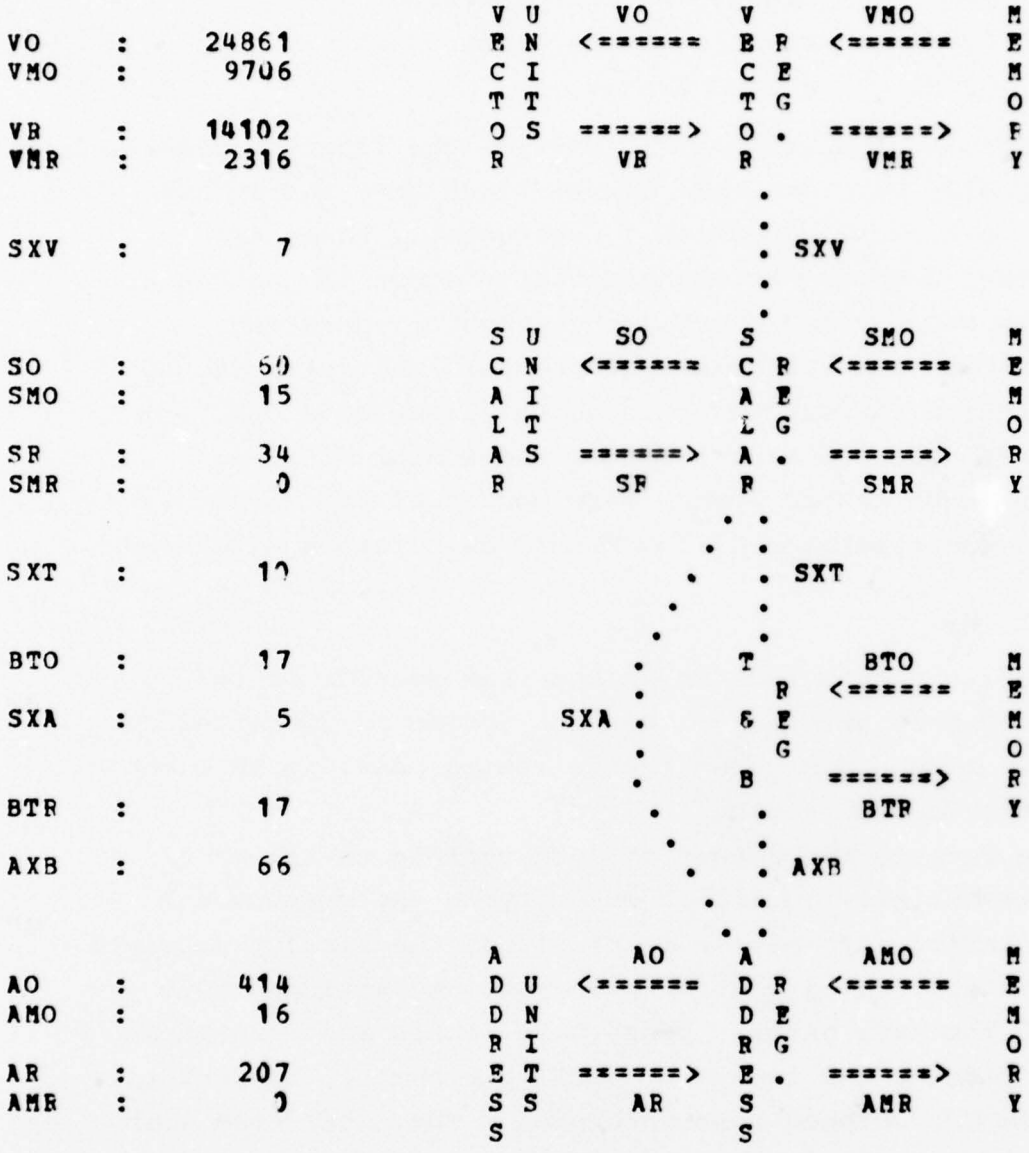
The Data Traffic Counts section is the last section of the STAT report. It is only printed if the FULL option is specified on the STAT command. Figure 2.5.3, on the following page, is an example of the Data Traffic Counts section.

This section reports the amount of data traffic on the major data trunks of the Cray-1. To aid in identifying the various data paths for which traffic information is provided, the simulator prints a block diagram of the Cray-1 central processor and attaches path labels to each of the data paths. These path labels are referenced on the left hand side of the report preceeding a number, representing the number of operands shipped over that path. The data paths with arrows are uni-directional whereas the paths shown dotted are bi-directional.

The left most column of the figure represents the Cray-1 computational units divided into three groups; vector, scalar and address. The floating point functional units are assumed to be shared between the vector and scalar groups.

The center column of the figure represents the Cray-1 register storage. Top to bottom these four register groups are the vector registers, the scalar registers, the T and B registers and the address registers. The vertical bi-directional communication paths (shown dotted)

DATA TRAFFIC COUNTS



MISC. : 237
BRANCHES : 4
FETCHES : 20
ISSUES : 1020

PERCENT OF VECTOR OPERANDS SUPPLIED BY CACHE - 60.96%
PERCENT OF TOTAL VECTOR TRAFFIC SUPPLIED BY CACHE - 69.15%
PERCENT VECTOR RESULTS OF TOTAL RESULTS - 98.32%
PERCENT VECTOR MEMORY TRAFFIC OF TOTAL MEMORY TRAFFIC - 96.78%
RATIO OF COMPUTATION TRAFFIC TO MEMORY TRAFFIC - 3.28
RATIO OF VECTOR MEMORY OPERANDS TO VECTOR MEMORY RESULTS - 4.19
RATIO OF VECTOR UNIT RESULTS TO VECTOR MEMORY OPERANDS - 1.45
RATIO OF VECTOR UNIT RESULTS TO VECTOR MEMORY RESULTS - 6.09
RATIO OF VECTOR UNIT CACHE USE TO VECTOR MEMORY CACHE USE - 3.24
VECTOR MEMORY RESULT RATE - 0.1425 RESULTS/CP

Figure 2.5.3 - Data Traffic Counts Table

between the four register groups are used for inter-group data transfers.

The right hand column of the figure represents Cray-1 main memory. MEMORY is shown in four sections only for the purpose of the figure. Any register group may reference any location in Cray-1 main memory.

The labeling scheme is defined as follows:

1) "A" means address, "S" means scalar and "V" means vector.
2) "O" means operands and "R" means result.
3) "X" means a bi-directional data path
4) "M" means the path is a memory path used by the three register groups tied both to memory and a computational unit. The T and B registers communicate only with memory and other register groups.

For example, "SMO" is the operand data path to the scalar registers from memory, where "SO" is the operand data path to the scalar computation units from the scalar registers.

Below the data path portion of the report, four other statistics are printed:

1) MISC. represents the number of miscellaneous instructions executed by the program that do not move data across any of the paths shown in the figure and are not branch instructions. The instructions counted include: op-codes 2-4, 20-22, 40-43, 72-73.
2) BRANCHES represent the number of branch instructions executed by the program whether the branch is taken or not.
3) FETCHES represent the number of parcel buffer fetches incurred by the running program.
4) ISSUES represent the number of instructions issued by the running program.

The last part of the Data Traffic Counts section shows nine percentage and ratio calculations. Each of these are discussed below with their derivation.

1. Percent of vector operands supplied by cache.

   The term cache refers to the eight Cray-1 vector registers. This percentage reflects the dominance of the cache over memory in supplying vector operands to the vector units. It is defined as,

$$\frac{VO-VMO}{VO} * 100$$

2. Percent of total vector traffic supplied by cache.

   This percentage is similar to (1) above, but also includes the effect of the vector results data traffic. It is defined as,

$$\frac{VO-VMO + VR-VMR}{VO + VR} * 100$$

3. Percent vector results of total results.

   This percentage is a measure of the vector-scalar composition of the program's computation. This figure reflects the percentage of all results computed in vector mode. Because scalar and vector instructions can execute concurrently, this figure is <u>not</u> the percentage of <u>time</u> spent in vector mode. This figure is defined as,

$$\frac{VR}{VR + SR + AR} * 100$$

4. Percent vector memory traffic of total memory traffic.

   This percentage is a measure of the vector-scalar composition of the program's memory usage. This figure reflects the percentage of vector traffic to and from the main memory. It is defined as,

$$\frac{VMO + VMR}{TMDT + FETCH} * 100$$

FETCH is the number of memory words read into the instruction parcel buffers. TMDT is the total memory data traffic and is defined as,

$$TMDT = VMO+VMR + SMO+SMR + AMO+AMR + BTO+BTR$$

5. Ratio of computation traffic to memory traffic.

This ratio is a measure of the benefit provided by the register portion of the Cray-1 memory hierarchy in reducing the main memory data traffic. If this ratio was one there would be no benefit in having the registers, since register traffic equals memory traffic. Typically this ratio is in the range of two to five indicating that the registers provide a substantial reduction in main memory data traffic. This ratio is defined as,

$$\frac{VO+VR + SO+SR + AO+AR}{TMDT}$$

6. Ratio of vector memory operands to vector memory results.

This ratio is a measure of the average vector operand requirements of the program. This ratio combined with the vector memory result rate (see 10 below) and the algorithmic complexity of main memory usage (the computational lifetime of data in main memory) will allow the algorithm designer to determine the mass storage I/O data rates necessary to keep the vector arithmetic units constantly busy. This ratio is defined as,

$$\frac{VMO}{VMR}$$

7. Ratio of vector unit results to vector memory operands.

This ratio is a figure of merit of the average value of main memory operands in the computation. A value of two would imply that each main memory operand precipitates

two arithmetic operations. This ratio is defined as,

$$\frac{VR}{VMO}$$

8. Ratio of vector unit results to vector memory results.

   This ratio is similar to (5) above, but compares only vector result traffic. This reflects the average number of arithmetic operations necessary for the production of one stored result. The ratio is defined as,

$$\frac{VR}{VMR}$$

9. Ratio of vector unit cache use to memory unit cache use.

   Like (5) above, this ratio measures the benefit of main memory bandwidth reduction provided by the registers in the memory hierarchy. However, this ratio only compares the data traffic into and out of the cache. It is defined as,

$$\frac{VO + VR}{VMO + VMR}$$

10. Vector memory result rate.

    This is the rate at which the program sends vector results to main memory. This is useful for assessing mass storage I/O data rates as mentioned in (6) above.

$$\frac{VMR}{RTC}$$

The ratios and percentages above tended to emphasize the vector portion of the Cray-1. Clearly, similar figures could be calculated for the scalar and address portions of the machine. The raw data is provided as part of the Data Traffic Counts report so that the user may calculate scalar figures or devise other measures of algorithm-processor performance.

2.5.2 CPACT Report

The CPACT report produces a detailed clock period activity record of the Cray-1 machine state. This is a 132 column report* suitable only for printing on a line printer. Figure 2.5.4 on the following page shows the format of the report. Across the top of the report, the various column headings are devoted to the Cray-1 resources that may be called into use by a Cray-1 instruction. Time flows down the page with each clock period of simulation time producing an output record that describes the state of Cray-1 resources at that clock period. With vector instructions using long vector lengths, the machine resource state may remain unchanged for fifty or more clock periods, resulting in many identical CPACT output records. The COMPRESS option on the CPACT command (see section 4) may be used to suppress the printing of ten or more identical output records. This substantially reduces simulation cost and makes the CPACT report far more manageable. One line of compression dots are printed in place of the suppressed records.

The CPACT report is partitioned into the following 21 Cray-1 resource fields:

1. ST. - The machine state field.

   This field indicates the machine state at each clock period. Three possible entries are: (1) "IS", which means that an instruction is issuing at this clock period, (2) blank which means that no instruction will issue at this clock period, and, (3) "FE", which means a parcel buffer fetch sequence is initiated at this clock period.

2. TAG - The activity resource tag.

   At a clock period in which a new machine activity (instruction issue or fetch request) is initiated, the activity is assigned a one letter activity resource tag (A-Z, Ø-9) which is used in subsequent clock periods to identify the Cray-1 resources called into use by the initiated activity. When a conflict occurs in the demand

*A narrow version of the CPACT report may be requested for printing at a terminal. See the CPACT command description.

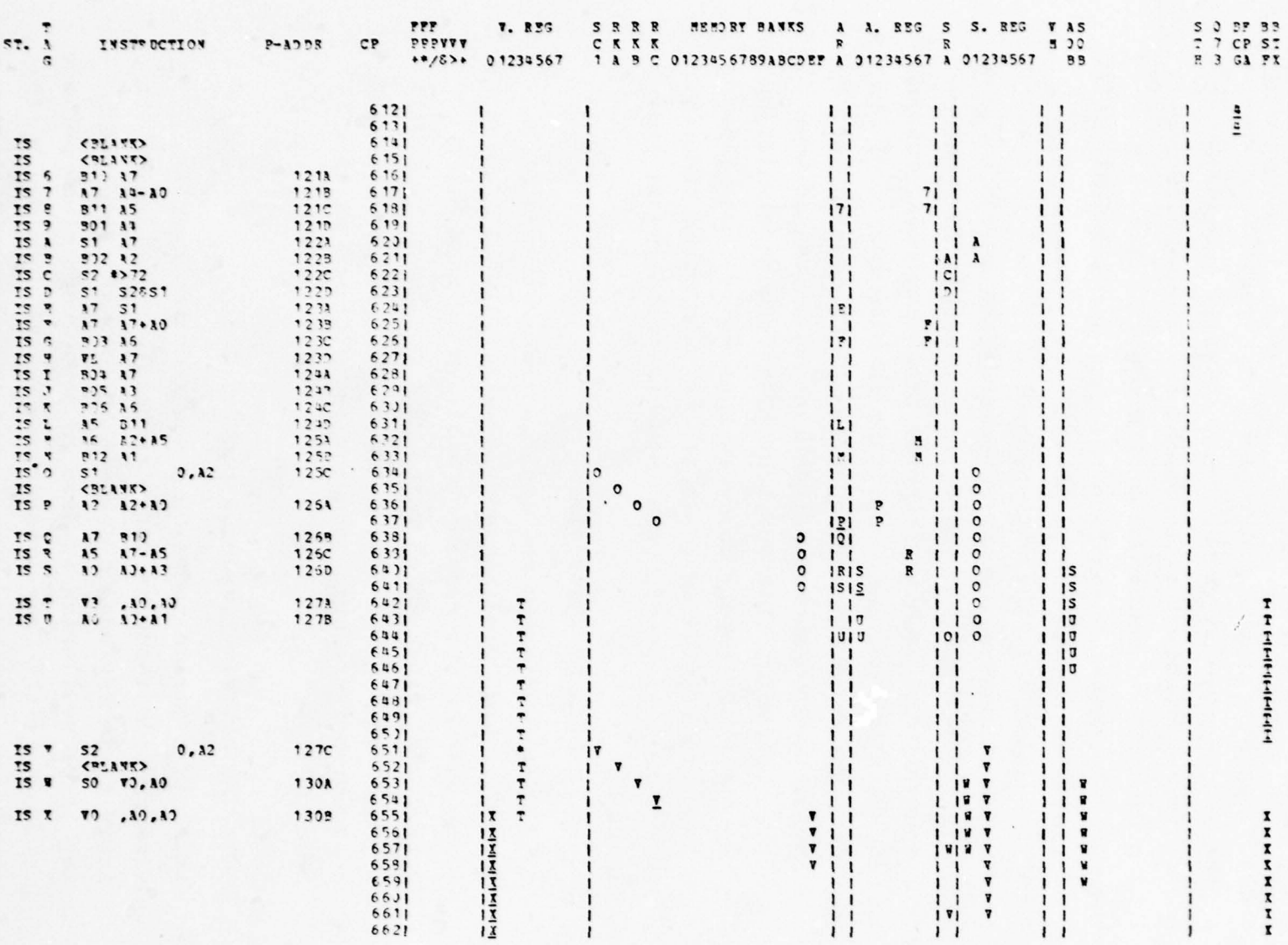


Figure 2.5.4 - CPACT Report Format

for a Cray-1 resource, the tag occupying the resource may be traced back to the initiating activity.

Resource conflict occurs when an activity initiated in a past clock period,occupies a Cray-1 resource that is now being demanded by another activity. For example, when an arithmetic instruction issues, the result register is reserved until the result arrives at the register. Because the Cray-1 is pipe-lined, a subsequent instruction, that requires the previous arithmetic result as an input operand, may experience an operand register conflict, causing it to hold issue until the previous arithmetic result arrives at the operand register (the previous instruction's result register). In this example the resource conflict occurs on a register. The CPACT report will show the first instruction's activity tag in the report column corresponding to the result register of the instruction. The tag will remain in this column until the result register reservation expires (i.e., the data has arrived). If the second instruction demands the use of this result register before the reservation has expired, the result register reservation tag will be underscored and the second instruction will hold issue until the data arrives.

The underscoring of activity tags is used throughout the report to highlight the resource conflicts of waiting instructions.

3. INSTRUCTION - The Mnemonic for the issuing instruction.

   When a Cray-1 instruction issues ("IS" in machine state field), the assembly mnemonic for the instruction is printed in this column.

4. P-ADDR - The parcel address of the issuing instruction.

   When a Cray-1 instruction issues, the parcel address from where it came is printed in this column.

5. CP - The simulator clock period.

   This column contains the simulator clock period. The clock period is reset to zero by an INIT command. If the user turns timing on and off through the SET command or the ERT, DRT instructions, the clock period is not affected but the machine resource state is cleared.

6. +*/¢>+ - The Cray-1 vector functional units.
   Each of these six columns represent the reservation state of a Cray-1 Vector functional unit. Left to right the units are : floating point adder, floating point multiplier, floating point reciprocal approximation, vector logical, vector shift, vector integer adder. The activity tag of a vector instruction which reserves one of these functional units will be placed in the corresponding column.

   The vector memory path can also be reserved by a vector instruction and is shown in one of the far right columns under the heading "BSF", which stands for block sequence flag. This flag is set during all vector memory references

7. V. Reg - The eight Cray-1 vector registers.

   Each of these eight columns represent the reservation state of a Cray-1 vector register. Vector registers are reserved by the vector instructions which reference them either as operand or result registers. The activity tag of the issuing vector instruction will be placed in the columns corresponding to the vector registers used by the instruction. Operand registers are typically reserved for MAX (VL,5) clock periods. Result registers are typically reserved for MAX(VL,5) +FUT + 2 clock periods, where FUT is the functional time of the vector unit performing the vector operation.

If a subsequent vector instruction requires, as an input vector, the result vector of a previous vector instruction, and is ready to issue when the prior instruction's first result arrives at it's vector register (the first result will arrive in FUT + 2 clock periods after issue), then the second vector instruction will issue only at the clock period when this first result arrives. This is called chaining and the clock period when the first result arrives is called chain slot time. If the second vector instruction misses chain slot time, it will hold issue until all results of the first instruction have arrived at the vector register.

If the second vector instruction chains to the first, the activity tag of the second instruction will replace the tag of the first instruction in the chained vector register column. If chain slot time is missed, an asterisk is placed in the result register field at the chain slot time clock period, highlighting chain slot time.

See appendix-A for a summary of Cray-1 timing information.

8. MEMORY BANKS - The Cray-1 rank registers and memory banks.

This portion represents the Cray-1 scalar memory reference access network and the 16 Cray-1 memory banks. The memory bank cycle time of the Cray-1 is four clock periods long. Consequently, memory accesses to the same bank must be at least four clock periods apart. Two scalar memory references which could address the same bank can issue two clock periods apart. This would give rise to a bank conflict which is resolved by the scalar rank register access network. (I/O access to main memory also passes through the rank registers). The four columns to the left of the 16 memory bank columns represent:

SC1 - Scalar in clock period one.
RKA - Rank register - A
RKB - Rank register - B
RKC - Rank register - C

When a scalar memory reference issues, it's activity tag is placed in the column SC1. It's bank address (lower 4 bits) is then compared to the bank addresses in rank registers A, B and C. If a bank coincidence is detected, the memory address waits at SC1 until a clock period arrives when bank coincidence vanishes. Meanwhile the bank addresses in the rank registers are advanced each clock period to the next rank register. The address in rank-C advances to it's target memory bank and remains latched at that bank for four clock periods. On the fifth clock period, the memory data is gated from the bank into the SEC-DED (single error correction - double error detection) network. Simultaneously a new memory address may be latched onto the bank to start the next reference.

While the address is waiting in SC1, the activity tag of the issuing instruction is placed in one of the far right columns labeled "STH", which means storage hold. While a scalar memory reference is waiting in storage hold, subsequent scalar memory references may not issue until the waiting scalar reference leaves the storage hold state. This means that two scalar memory reference instructions, accessing the same bank, may issue so as not to block subsequent instructions from issuing. But, if a third scalar memory reference tries to access the same bank as the first two references, the storage hold state of the second scalar reference will block issue of the third scalar reference which blocks all instruction issuing.

As the memory address advances through the rank registers, the activity tag of the issued memory reference is advanced to the right. When the tag leaves rank-C, it

will jump to its target bank and remain there for four clock periods.

Only scalar memory references place tags in this section of CPACT. Parcel buffer fetches, B and T-register transfers and vector references place no tags in this section. They do affect other columns of the report, though.

9. ARA - The A-register access path busy flag.

There is a single store access path to the eight Cray-1 address registers. Each clock period, one operand may be stored into one of the eight A-registers via this path. When an instruction tries to issue, if the result of its computation would make use of the A-register access path at a future clock period when the path is already reserved for use by a prior instruction, the issue will be held until the next clock period. When an instruction uses the access path its activity tag will appear for one clock period.

10. A. Reg - The eight Cray-1 address registers.

These eight columns correspond to the eight Cray-1 A-registers. When an instruction reserves an A-register its activity tag will appear in the appropriate column.

11. SRA - The S-register access path busy flag.

This flag serves the same function for the eight S-registers that the ARA flag does for the A-registers.

12. S.REG - The eight Cray-1 scalar registers.

These eight columns correspond to the eight Cray-1 S-registers. When an instruction reserves an S-register its activity tag will appear in the appropriate column.

13. VM - The vector mask busy flag.

    This flag is set when a 003 instruction (VM Sj) or a 175 instruction (VM Vj,C) is in progress. The activity tag of the issuing instruction appears in this column.

14. A0B - A0 busy flag.

    The A-register conditional branch instructions, 010-013, use the data in A0 to make branch decisions. When new data is stored A0 it takes two additional clock periods to validate the branch test flags. While the branch test flags are invalid, the A-register conditional branch instructions will hold issue. The activity tag of the instruction storing new data into A0 will appear in this column until the branch test flags are made valid.

15. S0B - S0 busy flag.

    The same comments for A0B above apply here, except the S-register conditional branch instructions (014-017) are affected.

16. STH - Storage hold flag.

    The activity tag for a scalar memory reference, whose address experiences a memory bank conflict with the rank registers is placed in this column until the conflict vanishes. Subsequent scalar memory references will hold issue until this flag clears. See the memory banks discussion for more details.

17. 073 - Vector mask read inhibit flag.

    Execution of a 003 instruction (VM Sj) or a 175 instruction (VM Vj,C) will cause a 073 instruction (Si VM) to hold issue until the 003 or 175 finishes. The activity tag of the 003 or 175 instruction will appear in this column.

18. BCG - The parcel buffer change flag.

When the next instruction parcel to enter the NIP (next instruction parcel) register is not in the current parcel buffer, due to a branch or a buffer fall through, this column will be tagged with an asterisk until the buffer change is completed. This may involve a switch to one of the other parcel buffers or a parcel buffer memory fetch may be needed. This flag causes all instructions to hold issue.

19. FPA - Fetch pause flag.

This flag is used to trigger a parcel buffer fetch sequence. While it is up, an asterisk appears in this column. The fetch sequence will begin when this flag is clear.

20. BSF - The vector memory reference block sequence flag.

When a vector memory reference (176,177) issues, this column will be set with the activity tag of the issuing instruction. Subsequent vector memory references will hold issue until this flag clears.

21. BTX - The B and T register block transfer flag.

When a B or T register block transfer instruction (034-037) issues, its activity flag will appear in this column. No other instruction may issue while a B or T block transfer is in progress.

2.5.3 CPACT Examples

To illustrate the use of the various CPACT report fields two examples are presented, a scalar memory reference example and a vector example.

2.5.3.1 Scalar Example

The CPACT report shown in figure 2.5.5 (wide and narrow versions) was programmed by the Cray-1 program below

```
20A:    S3      53,0      (B)
        A3      33,0      (C)
        A0      103,0     (D)
        S0      51,0      (E)
        S6      50,0      (F)
        S5      47,0      (G)
        JSZ     24C       (H)
24C:    EX      000       (I)
```

The labels to the left are the parcel addresses of the associated instructions. The letters in parentheses on the right, are the activity tags for the corresponding instruction as assigned by the CPACT report. These tags will be used to refer to the instructions in the discussion below.

When the program was first started up, no instruction parcels were in the parcel buffers so a fetch was required. The asterisk in the BCG field indicates a buffer change in process. When the first instruction parcel (20A) arrives at the parcel buffer, two pass instructions ("<BLANK>") issue which pull the parcel through the NIP to the CIP register. Instruction B issues at clock period 17 (CP 17) and its activity tag appears in SC1 to indicate a scalar memory reference in clock period one. As time advances the B tag propagates through the rank registers and onto memory bank 13 octal (B hexadecimal). After four clock periods on the memory bank, the B tag vanishes and appears later at clock period 27. Here the instruction makes use of the S-register access path so the memory data can be stored into the result register, S3 in this case.

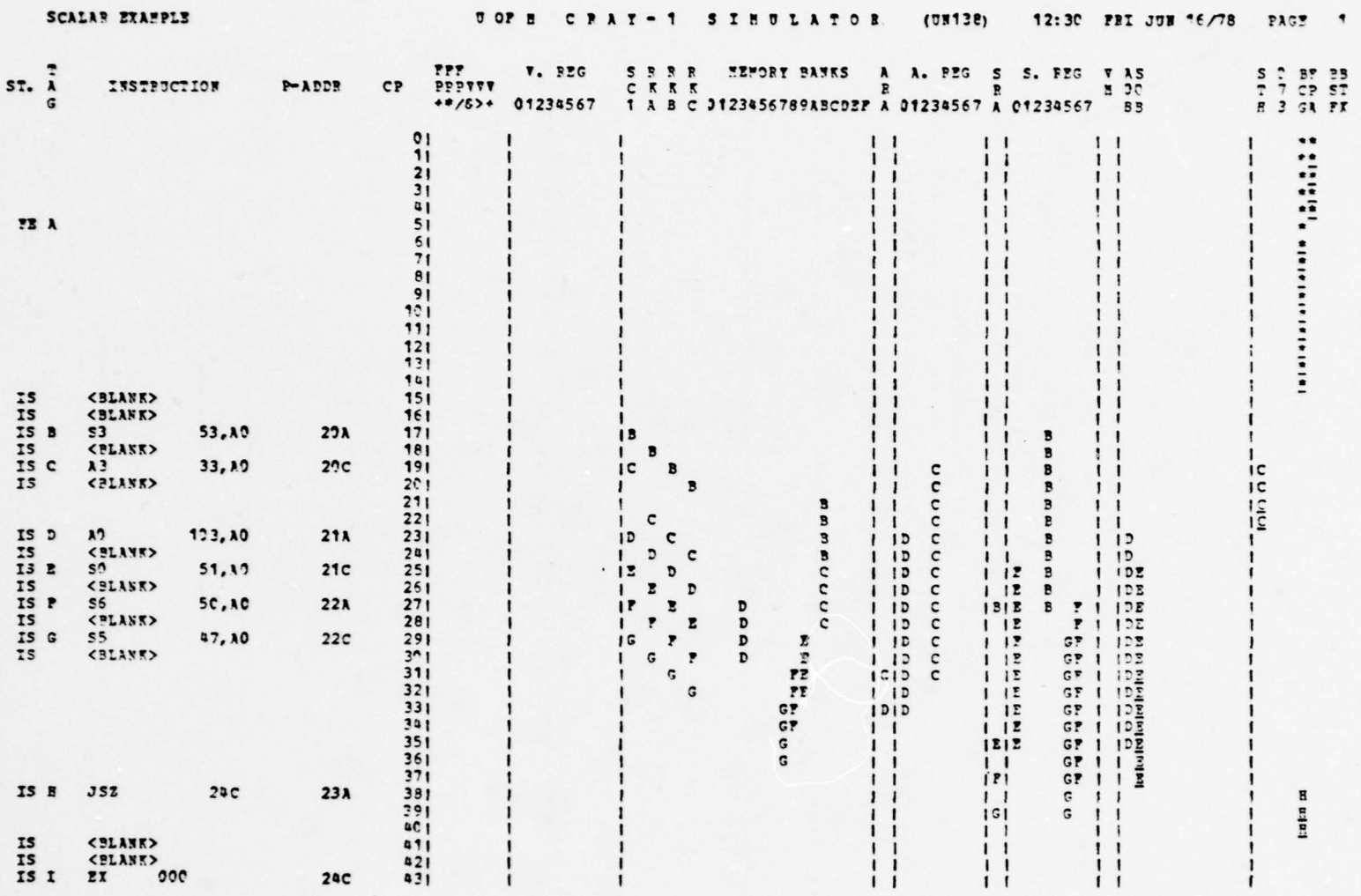


Figure 2.5.5 - CPACT Scalar Example, Wide version

```
                                                CRAY-1 SIMULATOR (UG088) AUG 22/78 PAGE  1

S T                                      FFF      V. REG  BSRRR S S. REG  A A. REG  S0BF
T A INSTRUCTION     P-ADDR    CP         PPPVVV           SCKKK R         R         T7CP
  G                                      +*/&>+ 01234567 F1ABC A01234567 A01234567 H3GA

                                        0|      |        |     |        |        |  **
                                        1|      |        |     |        |        |  **
                                        2|      |        |     |        |        |  **
                                        3|      |        |     |        |        |  **
                                        4|      |        |     |        |        |  **
F A                                     5|      |        |     |        |        |  *
                                        6|      |        |     |        |        |  *
                                        7|      |        |     |        |        |  *
                                        8|      |        |     |        |        |  *
                                        9|      |        |     |        |        |  *
                                       10|      |        |     |        |        |  *
                                       11|      |        |     |        |        |  *
                                       12|      |        |     |        |        |  *
                                       13|      |        |     |        |        |  *
                                       14|      |        |     |        |        |  *
I   <BLANK>                            15|      |        |     |        |        |
I   <BLANK>                            16|      |        |     |        |        |
I B S3      53,A0     20A              17|      |        | B   |     B  |        |
I   <BLANK>                            18|      |        |  B  |     B  |        |
I C A3      33,A0     20C              19|      |        | C B |     B  |     C  |C
I   <BLANK>                            20|      |        |    B|     B  |     C  |C
                                       21|      |        |     |     B  |     C  |C
                                       22|      |        |  C  |     B  |     C  |C
I D A0      103,A0    21A              23|      |        | D C |     B  |  D  C  |
I   <BLANK>                            24|      |        |  D C|     B  |  D  C  |
I E S0      51,A0     21C              25|      |        | E D | E   B  |  D  C  |
I   <BLANK>                            26|      |        |  E D| E   B  |  D  C  |
I F S6      50,A0     22A              27|      |        | F E |BE   B  F  |  D  C  |
I   <BLANK>                            28|      |        |  F E| E      F  |  D  C  |
I G S5      47,A0     22C              29|      |        | G F | E     GF  |  D  C  |
I   <BLANK>                            30|      |        |  G F| E     GF  |  D  C  |
                                       31|      |        |   G | E     GF  |CD  C  |
                                       32|      |        |    G| E     GF  | D      |
                                       33|      |        |     | E     GF  |DD      |
                                       34|      |        |     | E     GF  |        |
                                       35|      |        |     |EE     GF  |        |
                                       36|      |        |     |       GF  |        |
                                       37|      |        |     |F      GF  |        |
I H JSZ     24C       23A              38|      |        |     |       G   |        |  H
                                       39|      |        |     |G      G   |        |  H
                                       40|      |        |     |           |        |
I   <BLANK>                            41|      |        |     |           |        |
I   <BLANK>                            42|      |        |     |           |        |
EXIT    0 AT      24C
.
```

Figure 2.5.5 - CPACT Scalar Example, Narrow Version

During the time period from instruction issue (CP 27) until the data arrives at the result register (CP 27), the B tag appears in the S-register 3 column showing the reservation on S3. Because a scalar memory reference instruction is a two parcel instruction, a <BLANK> Will issue after it. This is true for all two parcel instructions.

When scalar reference instruction C issues, a bank conflict with the previous instruction is detected (address 33 and 53 are in the same bank). This causes the reference to enter the storage hold state until the conflict vanishes, meanwhile the C tag appears in the STH column. Scalar reference instruction D will try to issue at CP 21, but can't because the storage hold flag is set. This is noted by the underscore beneath the C tag in the STH column.

The subsequent four scalar references (D,E,F and G) all reference available memory banks and issue consecutively without conflict. Instruction E loads register S0 which is needed by branch instruction H to decide the branch outcome. Even though the data for E arrives at CP 35, two more clock periods are required until the branch condition flags become valid. While the branch flag is invalid the E tag appears in the S0 busy column (S0B). Once the S0B flag clears, the branch instruction issues at CP 38. This branch instruction has an in-buffer target address. While the buffer change is in progress, the instruction causing the change will place its tag in the BCG column. Once the change is complete, BCG is cleared and two blanks issue to load the CIP register.

2.5.3.2 Vector Example

This example illustrates the CPACT report with vector instructions. All vector lengths are seven. Figure 2.5.6 is the CPACT report generated by the following program:

```
20A:  A0    50      (B)
      A1    7       (C)
      VL    A1      (D)
      S1    43,0    (E)
      V0    A0,1    (F)
      V1    A0,1    (G)
      V2    V1+V0   (H)
      VM    V2,Z    (I)
      S0    VM      (J)
      JSN   40A     (K)
40A:  EX    000     (M)
```

As in the scalar example, the simulation begins with a fetch sequence. The first vector instruction (F) loads a vector from memory into V0. Rank-B and rank-C busy are hold issue conditions for this vector load and are shown underscored. Once the vector load issues, it places its tag both in the V0 busy and the block sequence flag (BSF) columns. Vector instruction G is also a vector memory load, but it must hold issue until BSF clears. The asterisk in the V0 column at clock period 33 represents the chain slot clock period for instruction F. If a vector instruction using V0 as an operand was placed after instruction F and it meets all other conditions at CP 33, it will issue at CP 33, and be chained to F. At CP 35 BSF clears, allowing instruction G (a second vector memory load) to issue.

Instruction H is a vector integer add with vector registers V1 and V0 as operands. Consequently, the busy state of V1 and V0 are hold issue conditions for H. The functional unit time for the vector load (G) is seven clock periods, so at CP 44 , (issue time) + (functional unit time) + (2), chain slot time will occur and the vector add issues. The tag for the chaining instruction (H) replaces the result tag (G) on the chained register.

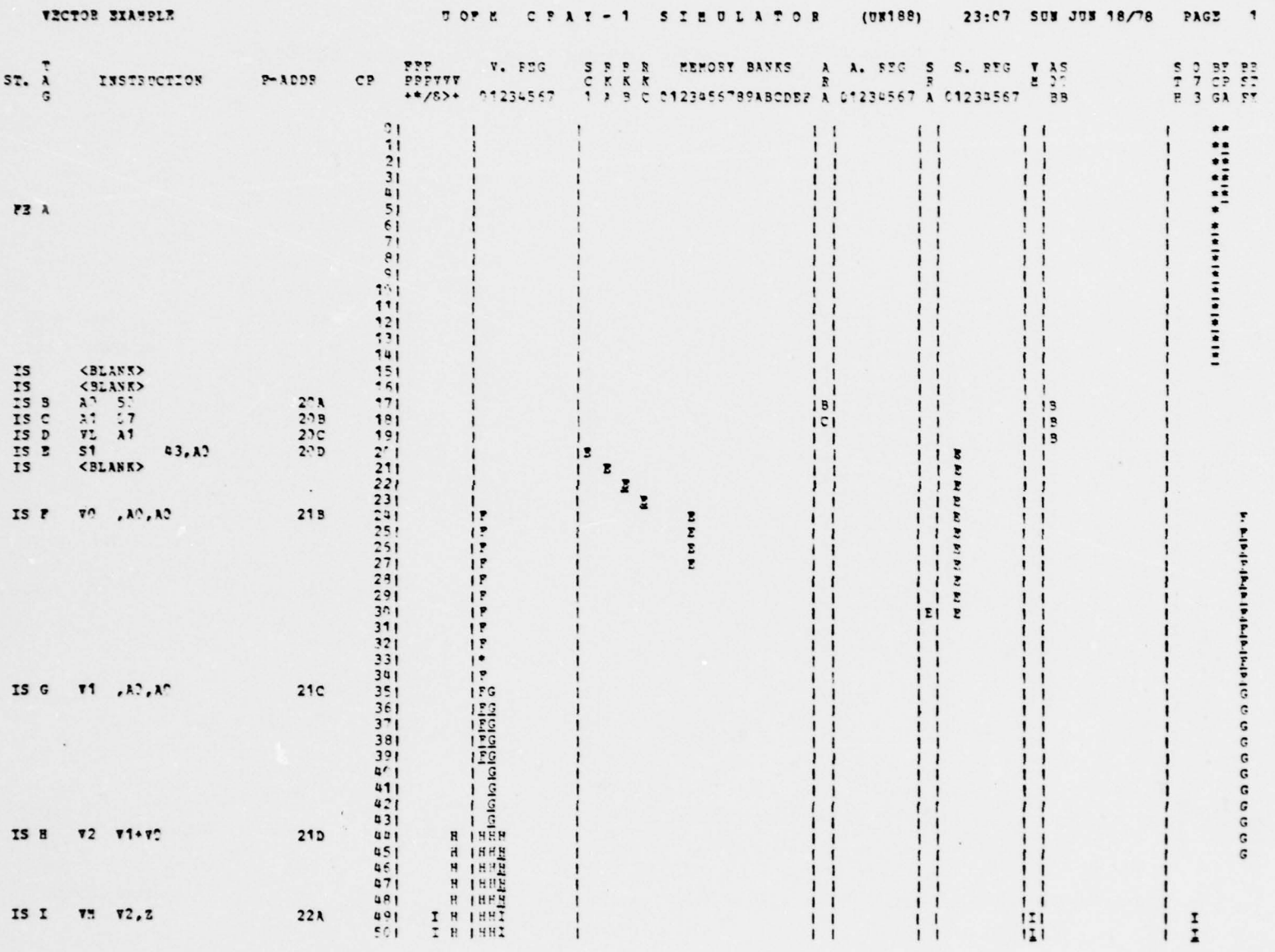


Figure 2.5.6 - CPACT Vector Example

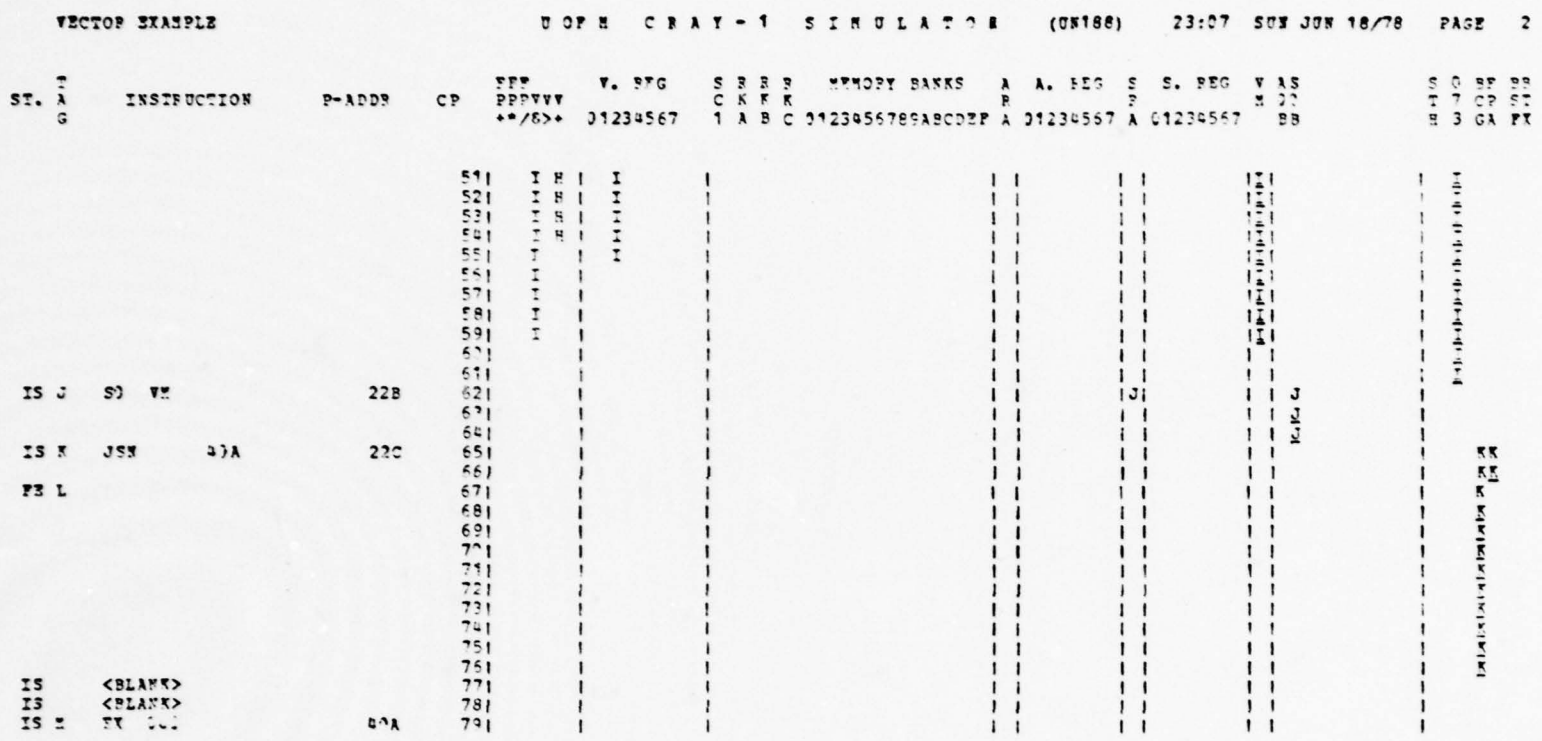


Figure 2.5.6 - CPACT Vector Example "continued"

Instruction I chains in a similar way to the result of instruction H. The vector mask register is reserved by I and its tag is placed in the VM column.

Instruction J will hold issue until the 073 inhibit flag clears. S0 is used as data for branch instruction K which does a branch out of buffer, causing a fetch sequence.

2.6 Inconsistencies with the Cray-1

In this section, simulator behavior that is known to be inconsistent with the Cray-1 will be discussed.

1) The simulator does not simulate Cray-1 I/O.
2) No Cray-1 monitor instructions are simulated.
3) The Cray-1 exchange mechanism is not simulated.
4) Recursive use of vector registers is not supported for all vector instructions. Recursive use is available for instructions 141, 143,147 and 171.
5) Cray-1 integer multiply of 64 bit operands is performed by the floating point multiply unit when the exponent field is found to be zero. This behavior is not currently supported in the Cray-1 simulator. Depending on the need for it, it may be added later.
6) Integers on the Cray-1 are 24 bits wide. Integers in the simulator are 32 bits wide.
7) The simulator floating point format (IBM 360/370) differs from the Cray-1 format. (See section 2.1.4)
8) The 071X2X instruction (Si +AK) produces a normalized result in the simulator. Not so on the Cray-1.
9) Though the timing of sizeable algorithms has been close to the Cray-1 timing, with an error in the 1/2% range, it is not exact. The source of this timing error is presently unknown.

3. Examples

This section is devoted to the discussion of two sample Cray-1 programs. The first example illustrates the Cray-1 simulator in standalone mode (no user main program) with a user supplied exit processor (CRAYEX). The second example illustrates calling the simulator by a user supplied main program.

3.1 Stand-alone use with an exit processor

This example illustrates the use of a user supplied exit processor. Entry to the exit processor is triggered by executing a Cray-1 exit instruction (004ijk, mnemonic EX ijk). As described in section 2.4, the Cray-1 simulator will call the subroutine CRAYEX with both the ijk field of the instruction and the Cray-1 registers as arguments. If the user supplies a CRAYEX subroutine, this routine will get control on all exit instructions which have non-zero ijk fields.

Appendix H shows a Fortran listing of an exit processor used to perform initialization and output. The Cray-1 program that uses this exit processor is a full matrix LU decomposition program.

When this program executes an "EX 1" instruction, CRAYEX gets control and initializes the full matrix which the program will decompose. Register A1 contains a pointer into the simulated Cray-1 memory where the matrix will start. Register A2 contains the size of the square matrix. The /MEMORY/ common block allows the CRAYEX routine to share the simulated Cray-1 memory. The user could equivalence other program arrays to the MEM array allowing convenient array sharing between the user Fortran program and the simulator memory.

When the program executes an "EX 2" instruction, CRAYEX gets control and prints both the matrix size and the run time. Register A7 contains the matrix size and register S7 contains the run time as sampled by the Cray-1 program. If the logical variable NOANS is set to .FALSE., the decomposed matrix will be printed. This is a way to interface the Cray-1 program with Fortran I/O.

A typical terminal session of this example is shown on the following page.

```
$RUN CRAYEX*CRAY1
*EXECUTION BEGINS
. LOAD FM.O
LOADING:
ADDRESS, LENGTH:  000034A 000450
. CHANGE M(34) D'5'
. DISPLAY M(34)
M(34)              = L'           0' R'           5'    0.0
. SET TIMING=ON MEMORY=PARITY
. RUN 35A #2000

 SIZE=     5  RTC=    1777
EXIT    2 AT       66D
. STAT



                              U OF M  CRAY-1  SIMULATOR   (UN138) SUN JUN 18,'78

                                   VECTOR  USAGE  COUNTS

CUM. TIMING      FP ADD  FP MUL  FP DIV    LOG.   SHIFT  I. ADD  V-LOAD  V-STOP

TIME BUSY (CP)       94     120       0       0       0       0     243     110
% TIME BUSY        5.18%   6.62%   0.0 %   0.0 %   0.0 %   0.0 %  13.40%   6.07%

NO. RESULTS          30      40       0       0       0       0      56      26
NO. VECTORS          16      20       0       0       0       0      27      11
AVERAGE VL         1.88    2.00     0.0     0.0     0.0     0.0    2.07    2.36

RUN TIME (CP) :    1813
MFLOPS        :    3.31
COMPOSITE AVL :    2.05
CONCURRENCY   :    0.31
MIPS          :   33.18

                              FLOATING  POINT  RESULT  COUNTS

                 ADDITION      MULTIPLICATION     RECIPROCAL          TOTAL

VECTOR (%)        30 ( 40.0)       40 ( 53.3)        0 (  0.0)       70 ( 93.3)
SCALAR (%)         0 (  0.0)        0 (  0.0)        5 (  6.7)        5 (  6.7)

TOTAL  (%)        30 ( 40.0)       40 ( 53.3)        5 (  6.7)       75 (100.0)

*STOP
*EXECUTION TERMINATED  T=.716      $.19
```

Terminal Session 3.1

4. Simulator Command Descriptions

This section describes each of the simulator commands in detail. Each command may be abbreviated and the minimum acceptable abbreviation is underlined. Each command description has the following format:

(1) Purpose - The function of the command.

(2) Prototype - The parameter syntax for the command.

(3) Description - A detailed description of the command.

(4) Examples.

(5) Error Responses - The possible error responses.

The following syntax is used to describe the commands:

Upper case characters must appear exactly as shown.

Lower case characters represent generic parameter names which must be replaced with the actual parameters.

Where blanks appear one or more blanks must appear.

Square brackets are used to denote optional parameters.

Ellipsis notation (...) is used to denote the repetition of a parameter list.

Vertical bars are used to separate parameter alternatives.

The following error responses apply to all commands:

Command length > 80 bytes -

All commands must be less than or equal to 80 characters in length. However, a simulator subroutine call may pass a segmented set of commands whose combined length may not exceed 200 bytes. Each individual command though may not exceed 80 bytes.

Command string terminator "!" not found -

On a simulator subroutine call the supplied command string did not contain an exclamation point within the first 200 bytes.

4. Simulator Command Descriptions

This section describes each of the simulator commands in detail. Each command may be abbreviated and the minimum acceptable abbreviation is underlined. Each command description has the following format:

(1) Purpose - The function of the command.

(2) Prototype - The parameter syntax for the command.

(3) Description - A detailed description of the command.

(4) Examples.

(5) Error Responses - The possible error responses.

The following syntax is used to describe the commands:

Upper case characters must appear exactly as shown.

Lower case characters represent generic parameter names which must be replaced with the actual parameters.

Where blanks appear one or more blanks must appear.

Square brackets are used to denote optional parameters.

Ellipsis notation (...) is used to denote the repetition of a parameter list.

Vertical bars are used to separate parameter alternatives.

The following error responses apply to all commands:

Command length > 80 bytes -

All commands must be less than or equal to 80 characters in length. However, a simulator subroutine call may pass a segmented set of commands whose combined length may not exceed 200 bytes. Each individual command though may not exceed 80 bytes.

Command string terminator "!" not found -

On a simulator subroutine call the supplied command string did not contain an exclamation point within the first 200 bytes.

Command Summary

AT	p-addr [skip-cnt]
BREAK	p-addr [skip-cnt]
CALCULATE	expression
CHANGE	symbol new-value
CLEAR	[p-addr ...]
COMMENT	any text
COST	
CPACT	[fdname [COMPRESS | NOCOMPRESS] [WIDE | NARROW]]
DEFINE	symbol constant[,w|,p|,v]
DISPLAY [@fmt-code]	symbol [,length]
DUMP	[module name]
ENDFILE	
HELP	command-name
IDENT	module-name
INIT	
LOAD	[s.a.] fdname ...
MAP	[XREF]
MTS	mts-command
REMOVE	symbol
RETURN	
RUN	[p-addr] [#issue-limit]
SET	lhs=rhs ...
STAT	[FULL]
STOP	
USE	fdname [NOECHO]
$MTS-command	

AT
Command Description

Purpose : To set an AT point at a selected parcel address

Prototype : AT p-addr [skip-count]
: commands to process when the AT point is hit.
END

Description:

An AT point is set at the specified parcel address. This address may be in modified octal format, which is the octal word address followed by a parcel code A,B,C, or D, or may be a symbol with parcel address attribute defined by the assembly language program. After the AT command is entered, the command language will read more command input. These commands are written, unprocessed, into a scratch file. To terminate input, enter the string "END" on a single line. AT points set at the lower parcel of a two parcel instruction are ignored.

When the AT point is hit, the scratch file will be opened and subsequent commands read from that file. The AT file is terminated with a RUN command which will resume the simulation automatically.

To regain control when the AT point is hit, you must enter the command "USE *MSOURCE*" when setting up the AT point.

An optional skip-count may be provided when first setting the AT point. This is a positive decimal number which indicates the number of times the simulator is to ignore the presence of this AT point. When this count expires, the AT point will be recognized and processed as above.

When the AT point is hit, the instruction at the p-addr, where it is set, will not have been executed.

Examples:

```
AT 21A
DISPLAY SØ AØ M(33)
END
```

When the AT point is hit, registers SØ, AØ and memory location 33 octal are displayed.

```
AT 245B 31
D M(Ø),1Ø
USE *MSOURCE*
END
```

An AT point is set at location 245B. The first 31 (decimal) times the instruction is executed, the AT point is skipped. Before the instruction is executed again the AT point takes control and displays memory locations through 10 (octal). Control is then given to the user terminal.

```
AT SUBRI
CHANGE A5 A7
END
```

This has the effect of patching a new instruction (A5 A7) at location with label SUBRI (i.e., A7 is stored into A5).

Error Responses:

Invalid p-addr -

The p-addr is unrecognizable or out of range of the current memory size.

Undefined Symbol -

The symbol specified is not defined in the current (IDENT) module.

Symbol does not reference a parcel address -

The symbol has word address or value attribute.

Invalid Skip count -

Skip count unrecognizable or negative.

No room for more Break or At points -

Only forty Break or At points may be set at any one time.

Break or At point already set here -

A Break or At point is already set at this p-addr.

Can't open AT point file -

The command language was unable to open the AT file for saving the commands. This could occur if the MTS scratch file character is changed from a minus sign.

BREAK

Command Description

Purpose : To control program flow by setting break points.

Prototype : BREAK p-addr [skip-cnt]

Description:

A break point is set at the indicated parcel address. The parcel address may be specified in a modified octal format: an octal word address followed by a parcel code A,B,C or D, or may be a symbol with parcel address attribute. An optional decimal skip count may be specified and will cause the break point to be ignored the indicated number of times.

The effect of hitting a break point is equivalent to issuing the command "USE *MSOURCE*". Continuation from a break point is accomplished by entering a RUN command. An end-of-file condition from the terminal (by $ENDFILE or control-c) will cause the command stack (see section 2.1.1) to be popped. This allows further commands to come from a prior USE command or a subroutine call command string. If a RUN command is entered without any parameters, the remaining issue limit is used and simulation continues with the broken instruction. No timing information is lost and no additional time is required. If a p-addr is specified on the RUN command, an instruction buffer fetch is forced and this will alter the timing.

When the break point is hit, the broken instruction has not been executed. Break points do not modify the Cray-1 memory, so they may be set before the program is loaded. A maximum of forty BREAK and AT points may be set. Break points on the lower parcel of a two parcel instruction are ignored. The user is not notified of this.

Examples:

```
BREAK  25A
BR     13B  18
B      LABI
```

Error Responses:

Invalid p-addr -

The p-addr is unrecognizable or out of range of the current memory size.

Invalid skip count -

Skip count unrecognizable or negative.

No room for more BREAK or AT points -

Only forty BREAK or AT points may be set at any one time.

Break or At point already set here -

A BREAK or AT point is already set at this p-addr.

Undefined Symbol -

The symbol specified is not defined in the current (IDENT) module.

Symbol does not reference a parcel address -

The symbol has word address or value attribute.

CALCULATE

Command description

Purpose : To calculate integer offsets for memory displacements

Prototype : CALCULATE <symbol><op><symbol>...<op><symbol>

Description:

The integer expression is evaluated strictly left to right. Only four operators (<op>) are allowed: *,+,-,/. The result is displayed in decimal, octal, and modified octal. The operands (<symbol>) may be replaced with any pre-defined or user defined symbol (see the DISPLAY command) or a constant as follows

nnn - for an octal integer constant
O'nnn'- for an octal integer constant
D'nnn'- for a decimal integer constant.

All operands are interpreted as integers with only the lower 32 bits of any 64 bit (e.g., S-registers) symbol taking part in the computation.

Examples:

```
CALC  O'131'+D'387'
CALC  A1*D'50'+ B.R1
CALC  M(32)+S4
```

Error Responses:

Calc unable to recognize operator -
Bad operator seen in expression.

Calc unable to recognize operand -
Bad operand or invalid pre-defined symbol seen in expression.

CHANGE

Command Description

Purpose : To alter Cray-1 storage locations in the simulator

Prototype : CHANGE symbol new-value

Description:

The CHANGE command allows any program accessible storage location in the Cray-1 simulator to be changed. The symbol parameter may be replaced with any of the predefined symbols which may be changed. See the DISPLAY command description for a list of the valid symbols.

The new-value parameter may be replaced with any pre-defined symbol or one of the following constants:

nnn - for an octal integer
O'nnn'- for an octal integer constant
D'nnn'- for a decimal integer constant
nnn.nnnDnn - for a double precision floating point constant.

Examples:

```
CHANGE A1 O'377'
CH      M(55) 2.5DØ
CH      V3(14) 2.32D27
CH      M(5Ø) M(1ØØ)
CH      B.BREG  T.TREG
CH      M(1) D'-1234567890123'
```

Error Responses:

Change unable to modify symbol -
See the DISPLAY command for a list of the symbols that may not be changed.

Change unable to evaluate symbol -
The symbol is not a legitimate symbol

CLEAR

Command description

Purpose : To clear break points and at points

Prototype : CLEAR [p-addr ...]

Description:

The CLEAR command is used to remove any break points or at points that have been previously set. If no parcel address parameters are specified, then all break and at points will be cleared. If one or more parcel addresses are supplied as arguments to the CLEAR command, then the break or at points set at these locations will be cleared.

Examples:

```
CLEAR
CL  21A  35C  74D  LOOP1
```

Error Responses:

Invalid p-addr -

The p-addr is invalid or out of range.

No break or at points are set -

Nothing set at this address -

Undefined Symbol -

The symbol specified is not defined in the current (IDENT) module.

Symbol does not reference a parcel address -

The symbol has word address or value attribute.

COMMENT

Command description

Purpose : To provide documentation about a simulator session

Prototype : COMMENT any text string

Description:

The COMMENT command is useful for documenting a terminal session or for generating advisory notices from AT command files or subroutine call-files. With AT command files, the commands are not echoed to the output device, however, COMMENT commands will echo. Also, on a subroutine call to the simulator COMMENT commands in the subroutine command string will echo regardless of the value of the echo parameter. Both of these features are useful to remind the user of any critical information.

Examples:

```
COMMENT  ANY TEXT STRING MAY BE SUPPLIED.
CO       THIS AT POINT ALTERS S3.
CO       DON'T FORGET TO SET UP LOCATION 34
```

Error Responses:

None

COST

Command description

Purpose : To print out processing costs.

Prototype : COST

Description:

The COST command will display simulator processing cost information. The cost figures cover the period from program start-up or the last INIT command to the present.

The following information is displayed:

```
SIMULATION COSTS SINCE LAST INIT -   TRISLV

RTC                    :          0
# INSTR. ISSUED       :        336
HOST CPU TIME          :          0.342 SEC
HOST COST              : $        0.074          LOW PRIORITY
PRINTING COSTS         : $        0.0    FOR     0 LINES    0 PAGES
INSTRUCTION RATE       :        982.5    INSTR./SEC
INSTRUCTION COST       :          0.220 $ / 1000 INSTR.
HOST / CRAY-1 TIME     :          0.0
```

1) The number of simulation clock periods.
2) The number of instructions issued.
3) The host (machine on which simulator is running) cpu time.
4) The host dollar cost and job priority.
5) The printing costs (useful for CPACT output).
6) The simulation rate in issued instructions per host cpu second.
7) The simulation cost in dollars per thousand issued instructions.
8) The ratio of host cpu time to the Cray-1 cpu time using the number of simulation clock periods multiplied by 12.5 ns.

Examples:

COST

Error Responses:

None.

CPACT
Command description

Purpose : To control the generation of the clock period activity report.

Prototype : CPACT [fdname [COMPRESS | NOCOMPRESS] [WIDE | NARROW]]

Description:

The CPACT command enables and disables the clock period activity report. If the fdname (MTS file or device name) is supplied, CPACT is enabled and the report is directed to the specified fdname. If no fdname is supplied CPACT is disabled. It should be noted that enabling CPACT will increase the simulation cost over the non-timing simulation mode by roughly a factor of forty to fifty. If timing is off (see the SET command) when CPACT is enabled, CPACT will turn TIMING on.

Since one line of output is generated for each clock period of simulation time, quite a bit of output can be generated fairly fast. To keep cost to a minimum, under MTS, CPACT should be sent directly to *PRINT*. If COMPRESS is specified, identical hold issue lines will be suppressed. COMPRESS is the default.

The normal CPACT report is suitable for printing on 132 column printers. If NARROW is specified or fdname corresponds to the user's terminal, the report will be condensed to 80 columns.

The report produced by CPACT is described in more detail in section 2.5.

Examples:

```
CPACT *PRINT*
CP
CP  *SINK*  NARROW
```

Error Responses:

CPACT already enabled -
CPACT with an fdname was given when CPACT was already enabled.

DEFINE
Command Description

Purpose : To define a new symbol

Prototype : DEFINE symbol constant[,W|,P|,V]

Description:

The DEFINE command adds a new symbol to the symbol table. It may then be used by other simulator commands.

The value of the symbol will be the constant. If the constant is octal, the type of the symbol will be word address. If the constant is modified octal, the type will be parcel address.

The default type may be overridden by appending ,W ,P or ,V to the constant. If this is done, the symbol will be defined as type word address, parcel address, or value, respectively.

Examples:

```
DEFINE   START   22B
DEF      ARAY1   100
DEF      BNAME   77,V
```

Error Responses:

Invalid constant

Invalid type -

type was not W,P, or V.

Duplicate symbol -

the symbol is already defined.

DISPLAY
Command Description

Purpose : To allow the user to examine the registers and memory of the simulated Cray-1.

Prototype : DISPLAY[@fmt] symbol[,length] ...

Description:

The DISPLAY command provides a faciltiy through which the user may examine Cray-1 registers and memory. The location to be displayed (symbol) is represented by any of the predefined symbols shown in the table below, or a user defined symbol. Subsequent contiguous locations can be displayed by providing a length parameter, separated from the symbol name by a comma. The length parameter may be a symbol name (e.g., VL) or a decimal integer constant. Also noted in this table is whether or not the CHANGE command will alter the symbol. DISPLAY output may be aborted by a keyboard attention.

Each symbol has a default display format associated with it. The default may be overridden for all symbols on the command by appending display format codes (fmt) to the command name. The format codes string is prefixed with "@". These format codes are defined as follows.

FORMAT		DISPLAY		
		64'	24'	16'
Code	Meaning	Operand	Operand	Operand
E	Floating pt.	Floating pt.	N.A.	N.A.
F	Fixed pt.	64' integer	24' integer	16' integer
O	Octal	64' octal	24' octal	16' octal
P	Parcel	4 octal parcels	N.A.	16' octal
M	Modified Octal	4 M. octal parcels	24' M. octal	16' M. octal
I	Instruction	4 Instr. Mnemonics	N.A.	Instr. Mnemonic
S	Symbolic	Symbol	Symbol	Symbol

User defined symbols are those symbols defined by the assembly language program and contained in a relocatable load module. These symbols may be one of three types: parcel address, word address or value. A parcel address symbol is treated as a 16 bit operand, and names a parcel

memory location. A word address symbol is treated as a 64 bit operand, and names a word memory location. A value symbol may be used to name an A,B,S,T or V register.

The only user defined symbols which may be referenced are those in the current module. See the IDENT command.

For the operand - format code combinations which are not applicable, no value will be displayed.

Pre-Defined Symbol Table

Symbol name	Cray-1 storage location	Region length allowed?	Change allowed?
Vn(elt)	Vector registers	Yes	Yes
Sn	Scalar registers	Yes	Yes
Tnn	T-registers	Yes	Yes
An	Address registers	Yes	Yes
Bnn	B-registers	Yes	Yes
M(addr)	Memory, words	Yes	Yes
IM(p-addr)	Memory, parcels	Yes	Yes
P	P-register	No	Yes
CIP	Current instruction parcel	No	No
NIP	Next instruction parcel	No	No
LIP	Lower instruction parcel	No	No
VM	Vector mask register	No	Yes
VL	Vector length register	No	Yes
RTC	Real time clock	No	Yes
XP	Exchange package	No	No
IBn(elt)	Instruction buffers	Yes	No

n or nn is a register number
elt is an element index within a register
addr is a word address, may be an expression
p-addr is a parcel address, may be an expression

Examples:

```
DISPLAY  A1 S3  A.LOOPCNTR  V.ROW1  SUBRTN1
D@O  AØ,8
D@PI  IM(p),1Ø MAIN, LEN$
D@EFO  VØ(Ø),VL
```

Error responses:

Invalid format code -

see format code list on page 64.

Invalid symbol

Invalid integer

DUMP

Command description

Purpose : To display the contents of all data areas of memory.

Prototype : DUMP [module-name]

Description:

The DUMP command displays the contents of memory addressed by all symbols of type word address. The memory locations are displayed in floating and fixed formats.

If a module-name is specified, only the module with corresponding IDENT name is dumped.

Examples:

```
DUMP
DU  SUBRI
```

Error Responses:

Module not loaded -

The specified module-name is not the name of any loaded module.

ENDFILE

Command description

Purpose : To signal an end-of-file condition to a USE command

Prototype : ENDFILE

Description:

This command terminates the effect of the current USE command. It pops the command stack causing input to be read from the previous source. See section 2.1.1 for more information about command input control. If a USE command is not in progress, ENDFILE is a no-op. That is, ENDFILE will not terminate a call-file or an AT-file.

Examples:

```
ENDFILE
E
```

Error Responses:

None.

HELP

Command description

Purpose : To provide on-line information about command syntax and function.

Prototype : HELP [cmd-name ...]

Description:

The HELP command takes as parameters the simulator command names. For each command name (cmd-name) given, a brief description is printed. A keyboard attention may be used to abort the HELP output. If no command name is provided a list of the legal commands is printed.

Examples:

```
HELP  DISPLAY  CHANGE
HELP
H HELP
```

Error Responses:

I can't help you -

The file containing the HELP responses doesn't exist or couldn't be accessed.

Error during HELP -

An error occurred during I/O to the output device.

IDENT

Command description

Purpose : To determine the subset of user defined symbols which may be referenced by other commands.

Prototype : IDENT module-name

Description :

Relocatable modules loaded by the simulator contain the definitions of all symbols defined in the assembly language program. Since assembly programs can be assembled and loaded independently, these symbols may not be unique. Only those symbols defined within a single module may be used at any given time.

The name field on the IDENT command must be the name contained on the IDENT record of one of the loaded modules. Only the symbols contained in that module will be available for use by other commands.

Examples:

```
IDENT MAIN
ID  SUBROUTN
```

Error Responses:

Module not loaded -

The name specified did not appear on the IDENT card of any loaded module.

INIT

Command description

Purpose : To re-initialize the simulator

Prototype : INIT

Description:

The INIT command allows re-initialization of the simulator state between runs of a program. It has the following effects:

1. All timing information is initialized.
2. All report information in initialized.
3. The simulator state is cleared.
4. The CPU clock is cleared.
5. The CIP, NIP and instruction buffers are invalidated.

INIT will not alter the A,B,S,T,V,VL,MODE, P and VM registers or memory

Examples

```
INIT
I
```

Error Responses:

None.

LOAD
Command description

Purpose : To load programs into the simulated CRAY-1 memory.

Prototype : LOAD [s.a.] fdname ...

Description:

The LOAD command opens the file or device (fdname) and reads one or more load modules from it. The load modules may be absolute or relocatable. See appendix K for a discussion of load module formats.

Absolute load modules are loaded at the address specified in the module. The octal starting address (s.a.), if specified, preceding the fdname, is ignored.

Relocatable modules are loaded at the first available 16 word boundary, unless an octal starting word address (s.a.) is specified. Modules will be relocated and linked by the loader.

Examples:

```
LOAD  OBJ
LOAD  30 FILE1
LOAD  *SOURCE*
```

Error responses

Unresolved externals exist -

A relocatable module references a module which is not loaded. The user will be prompted for more loader input.

MAP
Command Description

Purpose : To display the locations of all loaded modules

Prototype : MAP [XREF]

Description :

The MAP command will display the names of all loaded modules and their true addresses in CRAY-1 memory.

If XREF is specified, a two part cross reference table will be produced. The first part lists each module and the other modules it references. The second part lists each module and all modules which reference it.

Examples:

```
MAP
MAP  XREF
MA   X
```

Error responses:

None.

MTS

Command description

Purpose : To provide a command interface between the simulator and MTS.

Prototype : 1. MTS [mts-command]
2. $mts-command

Description:

The MTS command allows the user to pass commands to MTS without stopping the simulator. In the first prototype an optional MTS command may be supplied. Return is made to MTS with MTS processing the command. The user may restart the simulator with the $RESTART MTS command. The second prototype allows the issuing of a one-shot MTS command. That is, the command is passed to MTS but control returns to the simulator automatically when the command finishes. Any command input to the simulator that is prefixed with a dollar sign is treated as one-shot MTS command.

Examples:

```
MTS
M  DIS VMSIZE
$EMPTY -RPT
$EDIT  TRIDEC
$SDS
```

Error Responses:

None.

REMOVE

Command Description

Purpose : To remove a symbol from the simulator's symbol table.

Prototype : REMOVE symbol

Description:

The symbol specified is removed from the simulator tables. No further command language references to it will be allowed, unless it is redefined.

Examples:

```
REMOVE   START
REM      ARRAY1
```

Error Response:

symbol is not defined.

RETURN

Command description

Purpose : To allow the simulator to return to it's caller.

Prototype : RETURN

Description:

The RETURN command is used to force the simulator to return to it's caller. Normally, when the simulator is called as a subroutine, the CRAY1 interface subroutine will automatically place a RETURN command at the end of the call-file after all other user commands. As the call-file is processed this RETURN will eventually be executed. To cause an early return to the caller, the user may issue a RETURN command, thereby skipping the remaining commands in the call-file.

A RETURN command issued when running the simulator stand-alone is equivalent to a STOP command.

Examples:

```
RETURN
```

Error Responses:

None.

RUN

Command description

Purpose : To begin simulation of a Cray-1 program

Prototype : RUN [p-addr] [#issue-limit]

Description:

The RUN command is the only simulator command that actually begins the simulation.

The optional starting parcel address (p-addr) is the modified octal address, or symbol with parcel address attribute, of the first instruction to execute. If no p-addr is supplied, execution begins with the parcel pointed to by the Cray-1 P address register. If a p-addr is specified, an instruction buffer fetch sequence will always be forced even if the instruction is already in a parcel buffer. If no p-addr is supplied, a buffer fetch sequence will occur only if the P register points to an out of buffer instruction. (Note: changing the P register by the CHANGE command always forces a buffer fetch sequence.) Consequently, if no p-addr is given the program timing will be unaffected by whatever previously halted the simulation (a break point, an attention, etc.).

The optional issue limit parameter can be used to control the simulation by limiting the number of instructions that may issue. This must be a positive decimal number prefixed by a pound sign and is used to prevent run away programs or to allow single stepping through a program. If an issue limit is not provided the remainder of a previous issue limit is used or if no remainder is left, a default value of 1000 is supplied.

A single keyboard attention may be issued to stop the simulation. The simulation will be halted in such a way that no information is lost. The issue of a RUN command will continue the simulation as if it had never been interrupted.

Two keyboard attentions may be issued to exit to MTS if a single one doesn't stop the simulation. This could occur if either the simulator or a CRAYEX routine were looping. See section 2.4 about the CRAYEX dispatcher.

There are many conditions that can arise to stop the simulation. Normally, a run command will terminate when an EX instruction (004000) is executed and this is the usual way to stop a program. Other common conditions that stop simulation are break points, at points, issue limit expired or a keyboard attention. The exceptional conditions that halt simulation are discussed in section 2.2

Examples:

```
RUN  23C  #5000
RUN
R    MAINPROG
R#1  - to single step.
```

Error Responses:

```
RUN  unable to evaluate expression -
     Either the p-addr or the issue limit was invalid.
     The offending parameter is printed.
```

SET
Command Description

Purpose : To permit alteration of user setable switches

Prototype : SET lhs=rhs

Description:

The SET command allows the user to control some of the features of the simulator. Each parameter is composed of a left hand side (lhs), an equal sign and a right hand side (rhs). The left hand sides are the keyword names and the right hand sides are the new keyword values. The table below lists the legitimate left hand sides followed by a discussion of each one.

Keyword	Keyword values	Default
EFI	ON, OFF	ON
ISLIMIT	positive integer	1000
MACHINE	CRAY1, CRAY1-A	CRAY1
MEMORY	SECDED, PARITY	SECDED
OUTPUT	any fdname	*MSINK*
TIMING	ON, OFF	OFF
TITLE	a character string not exceeding 35 bytes	blank

EFI default: ON

The EFI (enable floating point interrupt) keyword allows user control of interrupts caused by:

1) Exponent overflow
2) Exponent underflow
3) Floating point division by zero.

If EFI is ON, the above three conditions will stop simulation. If EFI is OFF, these three conditions will be ignored when they occur. When the simulator starts up EFI is on by default. EFI is a mode bit in the Cray-1 mode register and the Cray-1 instructions EFI and DFI can also set or clear this bit.

ISLIMIT Default: 1000

The ISLIMIT keyword allows the user to change the default instruction issue limit. If no issue limit is specified on the RUN command and no remaining issue limit exists from previous run commands, the default instruction issue limit is used. A positive decimal integer must be specified on the right hand side. When the simulator starts up this keyword has a default value of 1000. Setting ISLIMIT to one is useful for single stepping through the program.

MACHINE Default: CRAY1

This keyword is intended for selecting the use of experimental architectural modifications to the Cray-1 simulator. The current legitimate keyword values are "CRAY1" and "CRAY1-A", with default being "CRAY1". When CRAY1 is selected, normal Cray-1 timing is in force. Currently, selecting CRAY1-A invokes only one Cray-1 architectural modification: that of improved memory bandwidth. With CRAY1-A, the data rates (in words per clock period) for block transfers (instructions 034-037, 176, 177) to and from main memory are shown in the table below. These data rates are a function of the address increment (K) used by the block transfer (one for 034-037, (Ak) for 176, 177).

K mod 16	Data Rate (wds/cp)	K mod 16	Data Rate (wds/cp)
0	.25	8	.5
1	4	9	4
2	2	10	2
3	4	11	4
4	1	12	1
5	4	13	4
6	2	14	2
7	4	15	4

When CRAY1-A is selected, chaining a vector arithmetic instruction off of a vector memory load (176) is disallowed. This is because of the possible imbalance in data rates between the two instructions. In general, this should not be a hardship since a reordering of vector instructions usually allows one to stagger the vector memory references to run in parallel with the arithmetic vector instructions.

MEMORY Default: SECDED

The first Cray-1 built by Cray Research Inc. has a memory which is protected by parity checking only. This was later found to be unsatisfactory and subsequent machines were built with SEC-DED (single error correction - double error detection) memory protection. By introducing SEC-DED on the memory, the access path to memory is one clock period longer than on the parity checked memory. This timing difference is user selectable in the simulator. By setting MEMORY to the value PARITY (e.g., SET MEMORY=PARITY), timing with the parity checked memory is possible. When the simulator starts up the default memory timing is SECDED.

OUTPUT Default: *MSINK*

The OUTPUT keyword controls the file or device to which the simulator sends all normal output (i.e., not prompts or error messages, which always go to the terminal). Normal output includes informational messages, DISPLAY, HELP, and STAT output. When the simulator starts up OUTPUT is set to *MSINK* (the terminal). With the SET command OUTPUT may be set to another file or device. A keyboard attention will switch the output back to *MSINK* automatically.

TIMING Default: OFF

The TIMING keyword controls simulator resource timing. If TIMING is off, only the results of instruction execution are computed. If TIMING is on, resource timing, reservation and issue constraints are simulated. By default, TIMING is off when the simulator starts up. Setting TIMING on increases the simulation cost by a factor of eight to ten. TIMING may also be enabled and disabled by the ERT and DRT instructions respectively. See section 5 for more explanation on ERT and DRT. Timing must be enabled to produce the CPACT report. However, the enabled or disabled state of CPACT is independent of the setting of TIMING. That is, turning TIMING on and off won't affect the enabled state of CPACT. However, the CPACT report is not producted while TIMING is disabled, but it will be resumed when TIMING is turned back on.

TITLE Default: blank

The TITLE keyword allows changing the title field on the CPACT and STAT reports. This title is always set when a LOAD command processes the title field on the load module header record. If no title field is provided TITLE is set to blanks. The right hand side of the TITLE keyword may be a string up to 35 characters long. No delimiters are required and the string may contain embedded blanks. All 35 characters will appear on the CPACT report but only the first 25 characters are printed on the STAT report.

Examples:

```
SET TIMING=ON   OUTPUT=-FILE   EFI=OFF
SET TITLE=THE CPACT TITLE   TIM=OFF   OUT=*PRINT*
SET MEMORY=PARITY   ISLIMIT=2500   MACHINE=CRAY1-A
```

Error Responses:

No equal sign found -

Invalid left hand side -

The keyword on the left hand side is unrecognizable.

Invalid right hand side -

The right hand side must be among the possible right hand sides for a given left hand side.

SET will have processed all keyword assignments up to the point of the error. An error aborts the rest of the SET command.

STAT

Command description

Purpose : To print out Cray-1 resource usage statistics

Prototype : STAT [FULL]

Description:

This command will print on the current output device a summary report of Cray-1 resource usage. This report is composed of the following three sections:

1. Vector Usage counts
2. Floating point result counts
3. Data traffic counts

The vector usage counts section is a timing measure of the program's vector use of the Cray-1 vector functional units. The data for this section is only collected when TIMING is ON. If TIMING is OFF when the STAT command is issued, this section will not be printed since, most likely, it would all be zero.

The floating point result counts section is a measure of the program's use of floating point computation. Floating point addition, multiplication and reciprocal operations are tabulated for both vector and scalar instructions. The data for this section is always collected regardless of the state of TIMING.

The data traffic counts section is a measure of the data (operands & results) flow throughout the Cray-1. Each major Cray-1 data path is illustrated on a figure, that is part of the report, along with the amount of traffic on each path. Also included in this section are some calculations of ratios and percentages based on the data traffic statistics. The formulas used for each calculation are printed beyond column 80 of the line containing the calculated number. Normally, these formulas won't appear on an 80 column terminal, but will be printed if the STAT output is diverted (via SET OUTPUT=*PRINT*) to the line printer.

The data for this section is always collected regardless of the state of TIMING. This section will not be printed unless the FULL option is specified on the stat command. The INIT command will reinitialize the STAT data collection.

This discussion is intended as a brief command description. For a detailed discussion of both the STAT and CPACT reports see section 2.5.

Examples:

```
STAT
STAT FULL
SET OUTPUT=*PRINT*
STAT
SET OUTPUT=*MSINK*
```

Error responses:

Extraneous parameter on STAT command -

This occurs if FULL is misspelled or improperly abbreviated.

STOP

Command description

Purpose : To terminate execution of the Cray-1 simulator.

Prototype : STOP

Description:

The STOP command terminates execution of the simulator, releases virtual memory used by the simulator and returns to MTS.

Examples:

```
STOP
```

Error Responses:

None.

USE

Command description

Purpose : To switch the command input stream to a file.

Prototype : USE fdname [NOECHO]

Description:

This command allows the user to put a long or frequently used command sequence in a file and have the simulator process those commands from that file. The fdname parameter is replaced with the name of an MTS file or device from which the simulator will read subsequent commands. Commands read from the file will automatically echo onto the current output device unless the optional NOECHO parameter is specified.

Any end-of-file condition or an ENDFILE command will terminate the USE command. This will pop the command stack causing input to resume with the previous source. The command stack is fifteen levels deep, allowing the user to nest USE commands as desired.

A keyboard attention may be used to abort any and all outstanding USE commands by resetting the command stack. This will cause the terminal to be current input device.

Examples:

```
USE   DISPFILE
USE  *MSOURCE* - to read from the terminal
U     CMDS  NOECHO
```

Error Responses:

USE command unable to open file -
The given fdname doesn't exist or access not allowed.

FDUB command stack overflow-
Attempt to nest more than 15 USE commands.

5. Cray - 1 Simulation Cost

This section will discuss the three different levels of simulation and compare their costs. The three simulation levels in increasing order of expense are:

(1) Non-timing simulation. Only results are computed. (TIMING = OFF)

(2) Timing simulation and result computation. (TIMING = ON)

(3) Resource activity trace simulation. (CPACT report)

To compare the relative costs of the different levels, we took a full matrix LU decomposition algorithm (with a 10 x 10 matrix) and simulated it at all three levels. In addition, we wrote a Fortran version and ran it directly on the host machine (an Amdahl 470/V6) used by the simulator. The results of the comparison are shown in Table 1.

The four comparison tests are:

Test - A: Fortran version compiled in IBM Fortran - H.

Test - B: Simulated Cray - 1 program, timing off.

Test - C: Simulated Cray - 1 program, timing on.

Test - D: Simulated Cray - 1 program, CPACT report generated.

The table column headings are left to right,

- The host cpu time in milliseconds.
- The host cpu time normalized to Test - B (fastest simulation mode).
- The host cpu time normalized to Test - A (Fortran version).
- The simulation instruction rate in instruction issues/second.
- The host cost in tenths of a cent.
- The host cost normalized to Test - B.
- The host cost normalized to Test - A.
- The simulation instruction cost in tenths of a cent per 1000 instructions.
- The ratio of the host cpu time to the predicted Cray - 1 cpu time. The Cray - 1 would run the problem approximately 13.4 times faster than the Amdahl Fortran version.

From the table it is seen that both the host cpu time and cost increases by a factor of 10 with timing enabled and increases by a factor of 30 when the resource activity trace (CPACT) is enabled.

	Test	Host CPU[2] time(ms)	Host CPU time χB	Host CPU time χA	Instruction rate (Issues/Sec)	Host cost[3] (mills)	Host Cost χB	Host Cost χA	Instruction Cost (mills/1000)	Host time to Cray-1 ti ratio
Fortran[1]	A	.950	-	1	-	.201	-	1	-	13.4
TIME=OFF	B	222	1	234	9,730	58	1	288	27	3,128
TIME=ON	C	2,159	9.7	2,273	1,000	569	9.8	2,831	263	30,424
CPACT	D	6,543	29.5	6,887	330	1,730[4]	29.8	8,607	801	92,903

2160 Instructions simulated
5677 Clock Periods Simulated
3.79 Average Vector Length

[1] IBM Fortran-H compiler

[2] Amdahl 470/V6

[3] Normal Priority, University/Govt. rates

[4] Plus $4.28 for printing costs (12,034 lines, 112 pages)

Table-1 Cray-1 Simulator Cost Comparison
For an LU Decomposition of a 10x10 Matrix

The ratio of simulation speed (on the Amdahl 470/V6) to Cray-1 speed is roughly:

3,100 to 1 with no timing,

30,500 to 1 with timing and,

92,200 to 1 with the activity trace.

With simulation on the Cray -1 these ratios would probably drop by a factor of three to five.

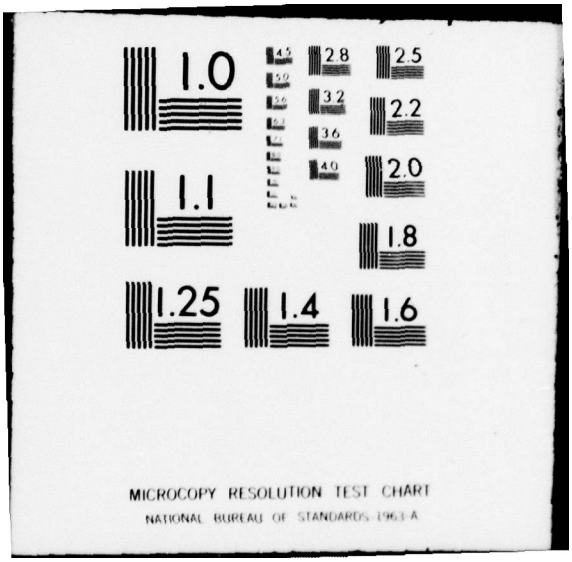
In addition to the SET command, which may turn timing on or off, two additional instructions ERT and DRT have been defined which also affect the simulator timing. ERT (002600) enables the CRAY-1 resource timing and is equivalent to the command SET TIMING=ON. DRT (002700) disables CRAY-1 resource timing and is equivalent to SET TIMING=OFF. To speed simulation, these instructions may be placed around code where timing is not desired.

AD-A060 638 MICHIGAN UNIV ANN ARBOR SYSTEMS ENGINEERING LAB F/G 9/2
A CRAY-1 SIMULATOR.(U)
SEP 78 D A ORBITS
UNCLASSIFIED SEL-118 AFOSR-TR-78-1405 AFOSR-75-2812 NL

2 OF 2
AD
A060638



END
DATE
FILMED
01-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

6. Bibliography

1) Cray-1 Reference Manual, publication no. 2240004, by Cray Research Inc., 1976.

2) Cray-1 Fortran Reference Manual, publication no. 2240009, by Cray Research Inc., 1977.

3) Cray-1 External Reference Specification, publication no. 2240011, by Cray Research Inc., 1976.

4) Cray-1 Fortran Mathematical Subprogram Library Reference Manual, publication no. 2240014, by Cray Research Inc., 1977

5) Cray-1 Reference Card, publication no. 2240003, by Cray Research Inc., 1977.

6) Cray-1 CAL Assembler Reference Manual, publication no. 2240000, 1976.

7) Introduction to Vector Processing on the Cray-1 Computer, publication no. 2240002, 1975.

8) The Cray-1 Computer System, CACM, by Richard M. Russel, Cray Research Inc., January 1978, pp. 63-72.

Appendix A.

Summary of Cray-1 Timing Information

This material has been borrowed from the Cray-1 Reference Manual, publication number 2240004, by Cray Research, Inc.

When issue conditions are satisfied an instruction completes in a fixed amount of time. Instruction issue may cause reservations to be placed on a functional unit or registers. Knowledge of the issue conditions, instruction execution times and reservations permit accurate timing of code sequences. Memory bank conflicts due to I/O activity are the only element of unpredictability.

SCALAR INSTRUCTIONS

Four conditions must be satisfied for issue of a scalar instruction:

1. The functional unit must be free. No conflicts can arise with other scalar instructions, however vector floating point instructions reserve the floating point units. Memory references may be delayed due to conflicts.
2. The result register must be free.
3. The operand register must be free.
4. Issue is delayed 1 clock period if a result register group input path conflict would exist with a previously issued instruction. One input path exists for each of the four register groups (A, B, S and T).

Scalar instructions place reservations only on result registers. A result register is reserved for the execution time of the instruction. No reservations are placed on the functional unit or operand registers.

A transmit scalar mask instruction to Si (073) instruction is delayed by (VL) + 6 clock periods from the issue of a previous vector mask (175) instruction, and is delayed by 6 clock periods from the issue of a preceding transmit (Sj) to VM (003) instruction.

Execution times in clock periods are given below. An asterisk indicates that issue may be delayed because of a functional unit reservation by a vector instruction. Memory may be considered a functional unit for timing considerations.

(A=A register, M=Memory, B=B register, S=S register, I=Immediate, C=Channel)

24-bit results:

A ← M	11*	A ← C	4
M ← A	1*	A ← A+A	2
A ← B	1	A ← AxA	6
B ← A	1	A ← pop(S)	4
A ← S	1	A ← lzc(S)	3
A ← I	1	VL ← A	1

64-bit results:

S ← M	11*	S ← S+S	3
M ← S	1*	S ← S(f.add)S	6*
S ← T	1	S ← S(f.mult)S	7*
T ← S	1	S ← S(r.a.)	14*
S ← I	1	S ← V	5
S ← S(log.)S	1	V ← S	3
S ← S(shift)I	2	S ← VM	1
S ← S(shift)A	3	S ← RTC	1
S ← S(mask)I	1	S ← A	2
RTC ← S	1	VM ← S	3

* Issue may be delayed because of a functional unit reservation by a vector instruction. Memory may be considered a functional unit for timing considerations.

VECTOR INSTRUCTIONS

Four conditions must be satisfied for issue of a vector instruction:

1. The functional unit must be free. (Conflicts may occur with vector operations.)
2. The result register must be free. (Conflicts may occur with vector operations.)
3. The operand registers must be free or at chain slot time.
4. Memory must be quiet if the instruction referemces memory.

Vector instructions place reservations on functional units and registers for the duration of execution.

1. Functional units are reserved for VL+4 clock periods. Memory is reserved for VL+5 clock periods on a write operation, VL+4 clock periods on a read operation.

2. The result register is reserved for the functional unit time +(VL+2) clock periods. The result register is reserved for the functional unit +7 clock periods if the vector length is less than 5. At functional unit time +2 (chain slot time) a subsequent instruction, which has met all other issue conditions, may issue. This process is called "chaining." Several instructions using different functional units may be chained in this manner to attain a significant enhancement of processing speed.

3. Vector operand registers are reserved for VL clock periods. Vector operand registers are reserved for 5 clock periods if the vector length is less than 5. The vector register used in a block store to memory (177 instruction) is reserved for VL clock periods. Scalar operand registers are not reserved.

Vector instructions produce one result per clock period. The functional unit times are given below. The vector read and write instructions (176, 177) produce results more slowly if bank conflicts arise due to the increment value (Ak) being a multiple of 8. Chaining cannot occur for the vector read or write operation in this case.

If (Ak) is an odd multiple of 8, results are produced every 2 clock periods.

If (Ak) is an even multiple of 8, results are produced every 4 clock periods.

Memory must be quiet before issue of the B and T register block copy instructions (034-037). Subsequent instructions may not issue for 14+(Ai) clock periods if (Ai)≠0 and 5 clock periods if (Ai)=0 when reading data to the B and T registers (034,036). They may not issue for 6+(Ai) clock periods when storing data (035,037).

The B and T register block read (034,036) instructions require that there be no register reservation on the A and S registers, respectively, before issue.

Branch instructions cannot issue until an A0 or S0 operand register has been free for one clock period. Fall-through in buffer requires two clock periods. Branch-in-buffer requires five clock periods. When an "out of buffer" condition occurs the execution time for a branch instruction is 14 clock periods.

A two parcel instruction takes two clock periods to issue.

Instruction issue is delayed 2 clock periods when the next instruction parcel is in a different instruction parcel buffer. Instruction issue is delayed 14 clock periods if the next instruction parcel is not in an instruction parcel buffer.

HOLD MEMORY

A delay of 1, 2, or 3 CP will be added to a scalar memory read if a bank conflict occurs with rank C, B, or A, respectively, of the memory access network. A conflict occurs if the address is in the same bank as the address in rank C, B, or A. Conflicts can occur only with scalar or I/O references. The scalar instruction senses the conflict condition at issue time + 1 CP. The scalar instruction address enters rank A of the memory access network at issue time + 1 CP. The scalar instruction address enters rank B at issue + 2 CP. The scalar instruction address enters rank C at issue + 3 CP.

Scalar instruction timing (no conflict):

```
CP n      Issue, reserve register
CP n+1    Address rank A, sense conflict
CP n+2    Address rank B
CP n+3    Address rank C
  :
CP n+9    Clear register reservation
CP n+10   Issue
```

HOLD ISSUE

A delay of issue results if a 100 - 137 instruction is in the CIP register and a hold memory condition exists. The delay will depend on the hold memory delay.

A delay of issue results if a 100 - 137 instruction is in the CIP register and a 100 - 137 instruction in process senses a conflict with rank A, B, or C.

An additional 1 CP delay is added to a hold memory condition if a 070 instruction conflict is sensed.

Appendix B.

Cray-1 Simulator I/O Device Usage

I/O in the Cray-1 simulator is done in two ways:

1) Through the use of standard Fortran data set reference numbers (DSRN) and,

2) Through the use of an MTS environment file or device usage block (FDUB).

The following I/O is done through DSRNs:

- All error messages use I/O unit Ø.
- All CPACT output uses I/O unit 1.
- All LOAD module input uses I/O unit 2.
- All normal Terminal output (echoing, etc.) uses I/O unit 3.
- All memory to memory I/O used for number conversion, etc., uses I/O unit 2Ø.

The following I/O is done through MTS provided FDUBs:

- All command input, whether from a call-file, an AT-file, a USE-file or the terminal is read using FDUBs. The command input stack is implemented with FDUBs.
- All HELP file responses are read from a file using a FDUB.
- The simulator driver tables are loaded at start up time using a FDUB.

The user should <u>not</u> use DSRNs Ø, 1, 2, 3 and 2Ø.

Appendix C.

Cray-1 Simulator Common Block Usage

The Cray-1 simulator currently uses 23 Fortran named common blocks. Except for /MEMORY/ and /MSIZE/ the user should not define symbols (subroutines or named common blocks) that conflict with common block names used by the simulator. These common block names are listed below:

ACTFLG	QCOM
BRKCOM	REG
COM$F	REPORT
CONTRL	SETABL
CTABLE	STATE
DECTBL	SYMTB1
DEV	SYMTB2
DRVTBL	TRKBLK
INSTRX	USAGE
MSFLAG	XCHANG
QCODES	$
LOAD	

Appendix D.

Establishing the Simulator on MTS

In addition to the object module which contains the Cray-1 simulator, three additional files and one initialization program are part of the simulator.

The initialization program (TABINIT) process the instruction driver table used by the simulator. TABINIT converts the driver table from a character format to an internal binary format which may be quickly read by the simulator when it starts up. This program is only needed if one changes the driver table.

The three additional files are:

1) OPFILE : The character format driver table used as input to TABINIT. (Not directly necessary to use the simulator.)
2) TABLES.DAT: The binary file which is output by TABINIT. This file is needed to run the simulator.
3) HELP : This file contains the help responses. It is not essential to use the simulator.

If TABLES.DAT and HELP are available under the CCID that is running the simulator, they will be read as they are needed.

Alternatively, one can recompile the subroutine OPFDUB (open fdub), after modifying the CCID defined in a DATA statement. This CCID should point to an alternate MTS catalog where TABLES.DAT and HELP can be found.

Appendix E.

Cray-1 Instruction Summary

This summary has been borrowed from the Cray-1 Reference Manual, publication number 2240004, by Cray Research, Inc.

CRAY-1	CAL		UNIT	DESCRIPTION
000000	ERR		-	Error exit
†000ijk	ERR	exp	-	Error exit
0010jk	CA,Aj	Ak	-	Set the channel (Aj) current address to (Ak) and begin the I/O sequence
0011jk	CL,Aj	Ak	-	Set the channel (Aj) limit address to (Ak)
0012j0	CI,Aj		-	Clear channel (Aj) interrupt flag
0013j0	XA	Aj	-	Enter XA register with (Aj)
0014j0	RT	Sj	-	Enter real-time clock register with (Sj)
00200k	VL	Ak	-	Transmit (Ak) to VL register
†002000	VL	1	-	Transmit 1 to VL register
002100	EFI		-	Enable interrupt on floating point error
002200	DFI		-	Disable interrupt on floating point error
0030j0	VM	Sj	-	Transmit (Sj) to VM register
†003000	VM	0	-	Clear VM register
004000	EX		-	Normal exit
†004ijk	EX	exp	-	Normal exit
0050jk	J	Bjk	-	Jump to (Bjk)
006ijkm	J	exp	-	Jump to exp
007ijkm	R	exp	-	Return jump to exp; set B00 to P
010ijkm	JAZ	exp	-	Branch to exp if (A0) = 0
011ijkm	JAN	exp	-	Branch to exp if (A0) ≠ 0
012ijkm	JAP	exp	-	Branch to exp if (A0) positive
013ijkm	JAM	exp	-	Branch to exp if (A0) negative
014ijkm	JSZ	exp	-	Branch to exp if (S0) = 0
015ijkm	JSN	exp	-	Branch to exp if (S0) ≠ 0
016ijkm	JSP	exp	-	Branch to exp if (S0) positive
017ijkm	JSM	exp	-	Branch to exp if (S0) negative
020ijkm	Ai	exp	-	Transmit jkm to Ai
021ijkm	Ai	exp	-	Transmit 1's complement of jkm to Ai
022ijk	Ai		-	Transmit jk to Ai
023ij0	Ai	Sj	-	Transmit (Sj) to Ai
024ijk	Ai	Bjk	-	Transmit (Bjk) to Ai
025ijk	Bjk	Ai	-	Transmit (Ai) to Bjk
026ij0	Ai	PSj	Pop/LZ	Population count of (Sj) to Ai
027ij0	Ai	ZSj	Pop/LZ	Leading zero count of (Sj) to Ai
030ijk	Ai	Aj+Ak	A Int Add	Integer sum of (Aj) and (Ak) to Ai
†030i0k	Ai	Ak	A Int Add	Transmit (Ak) to Ai
†030ij0	Ai	Aj+1	A Int Add	Integer sum of (Aj) and 1 to Ai
031ijk	Ai	Aj-Ak	A Int Add	Integer difference of (Aj) less (Ak) to Ai
†031i00	Ai	-1	A Int Add	Transmit -1 to Ai
031i0k	Ai	-Ak	A Int Add	Transmit the negative of (Ak) to Ai
031ij0	Ai	Aj-1	A Int Add	Integer difference of (Aj) less 1 to Ai
032ijk	Ai	Aj*Ak	A Int Mult	Integer product of (Aj) and (Ak) to Ai

CRAY-1	CAL		UNIT	DESCRIPTION
033i00	Ai	CI	-	Channel number to Ai (j=0)
033ij0	Ai	CA,Aj	-	Address of channel (Aj) to Ai (j≠0; k=0)
033ij1	Ai	CE,Aj	-	Error flag of channel (Aj) to Ai (j≠0; k=1)
034ijk	Bjk,Ai	,A0	Memory	Read (Ai) words at B register jk from (A0)
†034ijk	Bjk,Ai	0,A0	Memory	Read (Ai) words at B register jk from (A0)
035ijk	,A0	Bjk,Ai	Memory	Store (Ai) words at B register jk to (A0)
†035ijk	0,A0	Bjk,Ai	Memory	Store (Ai) words at B register jk to (A0)
036ijk	Tjk,Ai	,A0	Memory	Read (Ai) words at T register jk from (A0)
†036ijk	Tjk,Ai	0,A0	Memory	Read (Ai) words at T register jk from (A0)
037ijk	,A0	Tjk,Ai	Memory	Store (Ai) words at T register jk to (A0)
†037ijk	0,A0	Tjk,Ai	Memory	Store (Ai) words at T register jk to (A0)
040ijkm 041ijkm	Si	exp	- -	Transmit jkm to Si Transmit 1's complement of jkm to Si
042ijk	Si Si	<exp #>exp	S Logical	Form 1's mask in Si from the right
†042i77	Si	1	S Logical	Enter 1 into Si
043ijk	Si Si	<exp #>exp	S Logical	Form 1's mask in Si from the left
†043i00	Si	0	S Logical	Clear Si
044ijk	Si	Sj&Sk	S Logical	Logical product of (Sj) and (Sk) to Si
†044ij0	Si	Sj&SB	S Logical	Sign bit of (Sj) to Si
†044ij0	Si	SB&Sj	S Logical	Sign bit of (Sj) to Si (j≠0)
†045ijk	Si	#Sk&Sj	S Logical	Logical product of (Sj) and 1's complement of (Sk) to Si
†045ij0	Si	#SB&Sj	S Logical	(Sj) with sign bit cleared to Si
046ijk	Si	Sj\Sk	S Logical	Logical difference of (Sj) and (Sk) to Si
†046ij0	Si	Sj\SB	S Logical	Toggle sign bit of Sj, then enter into Si
†046ij0	Si	SB\Sj	S Logical	Toggle sign bit of Sj, then enter into Si (j≠0)
047ijk	Si	#Sj\Sk	S Logical	Logical equivalence of (Sk) and (Sj) to Si
†047i0k	Si	#Sk	S Logical	Transmit 1's complement of (Sk) to Si
†047ij0	Si	#Sj\SB	S Logical	Logical equivalence of (Sj) and sign bit to Si
†047ij0	Si	#SB\Sj	S Logical	Logical equivalence of (Sj) and sign bit to Si (j≠0)
†047i00	Si	#SB	S Logical	Enter 1's complement of sign bit into Si
050ijk	Si	Sj!Sj&Sk	S Logical	Logical product of (Si) and (Sk) complement ORed with logical product of (Sj) and (Sk) to Si
†050ij0	Si	Sj!Si&SB	S Logical	Scalar merge of (Sj) and sign bit of (Sj) to Si
051ijk	Si	Sj!Sk	S Logical	Logical sum of (Sj) and (Sk) to Si
†051i0k	Si	Sk	S Logical	Transmit (Sk) to Si
†051ij0	Si	Sj!SB	S Logical	Logical sum of (Sj) and sign bit to Si
†051ij0	Si	SB!Sj	S Logical	Logical sum of (Sj) and sign bit to Si (j≠0)
†051i00	Si	SB	S Logical	Enter sign bit into Si
052ijk	S0	Si<exp	S Shift	Shift (Si) left exp places to S0
053ijk	S0	Si>exp	S Shift	Shift (Si) right exp places to S0
054ijk	Si	Si<exp	S Shift	Shift (Si) left exp places
055ijk	Si	Si>exp	S Shift	Shift (Si) right exp places
056ijk	Si	Si,Sj<Ak	S Shift	Shift (Si and Sj) left (Ak) places to Si
†056ij0	Si	Si,Sj<1	S Shift	Shift (Si and Sj) left one place to Si
†056i0k	Si	Si<Ak	S Shift	Shift (Si) left (Ak) places to Si

CRAY-1	CAL		UNIT	DESCRIPTION
057ijk	Si	Sj,Si>Ak	S Shift	Shift (Sj and Si) right (Ak) places to Si
†057ij0	Si	Sj,Si>1	S Shift	Shift (Sj and Si) right one place to Si
†057i0k	Si	Si>Ak	S Shift	Shift (Si) right (Ak) places to Si
060ijk	Si	Sj+Sk	S Int Add	Integer sum of (Sj) and (Sk) to Si
061ijk	Si	Sj-Sk	S Int Add	Integer difference of (Sj) and (Sk) to Si
†061i0k	Si	-Sk	S Int Add	Transmit negative of (Sk) to Si
062ijk	Si	Sj+FSk	F.P. Add	Floating sum of (Sj) and (Sk) to Si
†062i0k	Si	+FSk	F.P. Add	Normalize (Sk) to Si
063ijk	Si	Sj-FSk	F.P. Add	Floating difference of (Sj) and (Sk) to Si
†063i0k	Si	-FSk	F.P. Add	Transmit normalized negative of (Sk) to Si
064ijk	Si	Sj*FSk	F.P. Mult	Floating product of (Sj) and (Sk) to Si
065ijk	Si	Sj*HSk	F.P. Mult	Half precision rounded floating product of (Sj) and (Sk) to Si
066ijk	Si	Sj*RSk	F.P. Mult	Full precision rounded floating product of (Sj) and (Sk) to Si
067ijk	Si	Sj*ISk	F.P. Mult	2 - Floating product of (Sj) and (Sk) to Si
070ij0	Si	/HSj	F.P. Rcpl	Floating reciprocal approximation of (Sj) to Si
071i0k	Si	Ak	-	Transmit (Ak) to Si with no sign extension
071i1k	Si	+Ak	-	Transmit (Ak) to Si with sign extension
071i2k	Si	+FAk	-	Transmit (Ak) to Si as unnormalized floating point number
071i30	Si	0.6	-	Transmit constant 0.75*2**48 to Si
071i40	Si	0.4	-	Transmit constant 0.5 to Si
071i50	Si	1.	-	Transmit constant 1.0 to Si
071i60	Si	2.	-	Transmit constant 2.0 to Si
071i70	Si	4.	-	Transmit constant 4.0 to Si
072i00	Si	RT	-	Transmit (RTC) to Si
073i00	Si	VM	-	Transmit (VM) to Si
074ijk	Si	Tjk	-	Transmit (Tjk) to Si
075ijk	Tjk	Si	-	Transmit (Si) to Tjk
076ijk	Si	Vj,Ak	-	Transmit (Vj, element (Ak)) to Si
077ijk	Vi,Ak	Sj	-	Transmit (Sj) to Vi element (Ak)
†077i0k	Vi,Ak	0	-	Clear Vi element (Ak)
10hijkm	Ai	exp,Ah	Memory	Read from ((Ah) + exp) to Ai (A0=0)
†100ijkm	Ai	exp,0	Memory	Read from (exp) to Ai
†100ijkm	Ai	exp,	Memory	Read from (exp) to Ai
†10hi000	Ai	,Ah	Memory	Read from (Ah) to Ai
11hijkm	exp,Ah	Ai	Memory	Store (Ai) to (Ah) + exp (A0=0)
†110ijkm	exp,0	Ai	Memory	Store (Ai) to exp
†110ijkm	exp,	Ai	Memory	Store (Ai) to exp
†11hi000	,Ah	Ai	Memory	Store (Ai) to (Ah)
12hijkm	Si	exp,Ah	Memory	Read from ((Ah) + exp) to Si (A0=0)
†120ijkm	Si	exp,0	Memory	Read from exp to Si
†120ijkm	Si	exp,	Memory	Read from exp to Si
†12hi000	Si	,Ah	Memory	Read from (Ah) to Si
13hijkm	exp,Ah	Si	Memory	Store (Si) to (Ah) + exp (A0=0)
†130ijkm	exp,0	Si	Memory	Store (Si) to exp
†130ijkm	exp,	Si	Memory	Store (Si) to exp
†13hi000	,Ah	Si	Memory	Store (Si) to (Ah)
140ijk	Vi	Sj&Vk	V Logical	Logical products of (Sj) and (Vk) to Vi
†140i00	Vi	0	V Logical	Clear Vi
141ijk	Vi	Vj&Vk	V Logical	Logical products of (Vj) and (Vk) to Vi

CRAY-1	CAL		UNIT	DESCRIPTION
142ijk	Vi	Sj!Vk	V Logical	Logical sums of (Sj) and (Vk) to Vi
†142i0k	Vi	Vk	V Logical	Transmit (Vk) to Vi
143ijk	Vi	Vj!Vk	V Logical	Logical sums of (Vj) and (Vk) to Vi
144ijk	Vi	Sj\Vk	V Logical	Logical differences of (Sj) and (Vk) to Vi
145ijk	Vi	Vj\Vk	V Logical	Logical differences of (Vj) and (Vk) to Vi
146ijk	Vi	Sj!Vk&VM	V Logical	Transmit (Sj) if VM bit = 1; (Vk) if VM bit = 0 to Vi
†146i0k	Vi	#VM&Vk	V Logical	Vector merge of (Vk) and 0 to Vi
147ijk	Vi	Vj!Vk&VM	V Logical	Transmit (Vj) if VM bit = 1; (Vk) if VM bit = 0 to Vi
150ijk	Vi	Vj<Ak	V Shift	Shift (Vj) left (Ak) places to Vi
†150ij0	Vi	Vj<1	V Shift	Shift (Vj) left one place to Vi
151ijk	Vi	Vj>Ak	V Shift	Shift (Vj) right (Ak) places to Vi
†151ij0	Vi	Vj>1	V Shift	Shift (Vj) right one place to Vi
152ijk	Vi	Vj,Vj<Ak	V Shift	Double shift (Vj) left (Ak) places to Vi
†152ij0	Vi	Vj,Vj<1	V Shift	Double shift (Vj) left one place to Vi
153ijk	Vi	Vj,Vj>Ak	V Shift	Double shift (Vj) right (Ak) places to Vi
153ij0	Vi	Vj,Vj>1	V Shift	Double shift (Vj) right one place to Vi
154ijk	Vi	Sj+Vk	V Int Add	Integer sums of (Sj) and (Vk) to Vi
155ijk	Vi	Vj+Vk	V Int Add	Integer sums of (Vj) and (Vk) to Vi
156ijk	Vi	Sj-Vk	V Int Add	Integer differences of (Sj) and (Vk) to Vi
†156i0k	Vi	-Vk	V Int Add	Transmit negative of (Vk) to Vi
157ijk	Vi	Vj-Vk	V Int Add	Integer differences of (Vj) and (Vk) to Vi
160ijk	Vi	Sj*FVk	F.P. Mult	Floating products of (Sj) and (Vk) to Vi
161ijk	Vi	Vj*FVk	F.P. Mult	Floating products of (Vj) and (Vk) to Vi
162ijk	Vi	Sj*HVk	F.P. Mult	Half precision rounded floating products of (Sj) and (Vk) to Vi
163ijk	Vi	Vj*HVk	F.P. Mult	Half precision rounded floating products of (Vj) and (Vk) to Vi
164ijk	Vi	Sj*RVk	F.P. Mult	Rounded floating products of (Sj) and (Vk) to Vi
165ijk	Vi	Vj*RVk	F.P. Mult	Rounded floating products of (Vj) and (Vk) to Vi
166ijk	Vi	Sj*IVk	F.P. Mult	2 - floating products of (Sj) and (Vk) to Vi
167ijk	Vi	Vj*IVk	F.P. Mult	2 - floating products of (Vj) and (Vk) to Vi
170ijk	Vi	Sj+FVk	F.P. Add	Floating sums of (Sj) and (Vk) to Vi
†170i0k	Vi	+FVk	F.P. Add	Normalize (Vk) to Vi
171ijk	Vi	Vj+FVk	F.P. Add	Floating sums of (Vj) and (Vk) to Vi
172ijk	Vi	Sj-FVk	F.P. Add	Floating differences of (Sj) and (Vk) to Vi
†172i0k	Vi	-FVk	F.P. Add	Transmit normalized negatives of (Vk) to Vi
173ijk	Vi	Vj-FVk	F.P. Add	Floating differences of (Vj) and (Vk) to Vi
174ij0	Vi	/HVj	F.P. Rcpl	Floating reciprocal approximations of (Vj) to Vi
1750j0	VM	Vj,Z	V Logical	VM=1 where (Vj) = 0
1750j1	VM	Vj,N	V Logical	VM=1 where (Vj) ≠ 0
1750j2	VM	Vj,P	V Logical	VM=1 where (Vj) positive
1750j3	VM	Vj,M	V Logical	VM=1 where (Vj) negative
176i0k	Vi	,A0,Ak	Memory	Read (VL) words to Vi from (A0) incremented by (Ak)
†176i00	Vi	,A0,1	Memory	Read (VL) words to Vi from (A0) incremented by 1
1770jk	,A0,Ak	Vj	Memory	Store (VL) words from Vj to (A0) incremented by (Ak)
†1770j0	,A0,1	Vj	Memory	Store (VL) words from Vj to (A0) incremented by 1

† Special syntax form

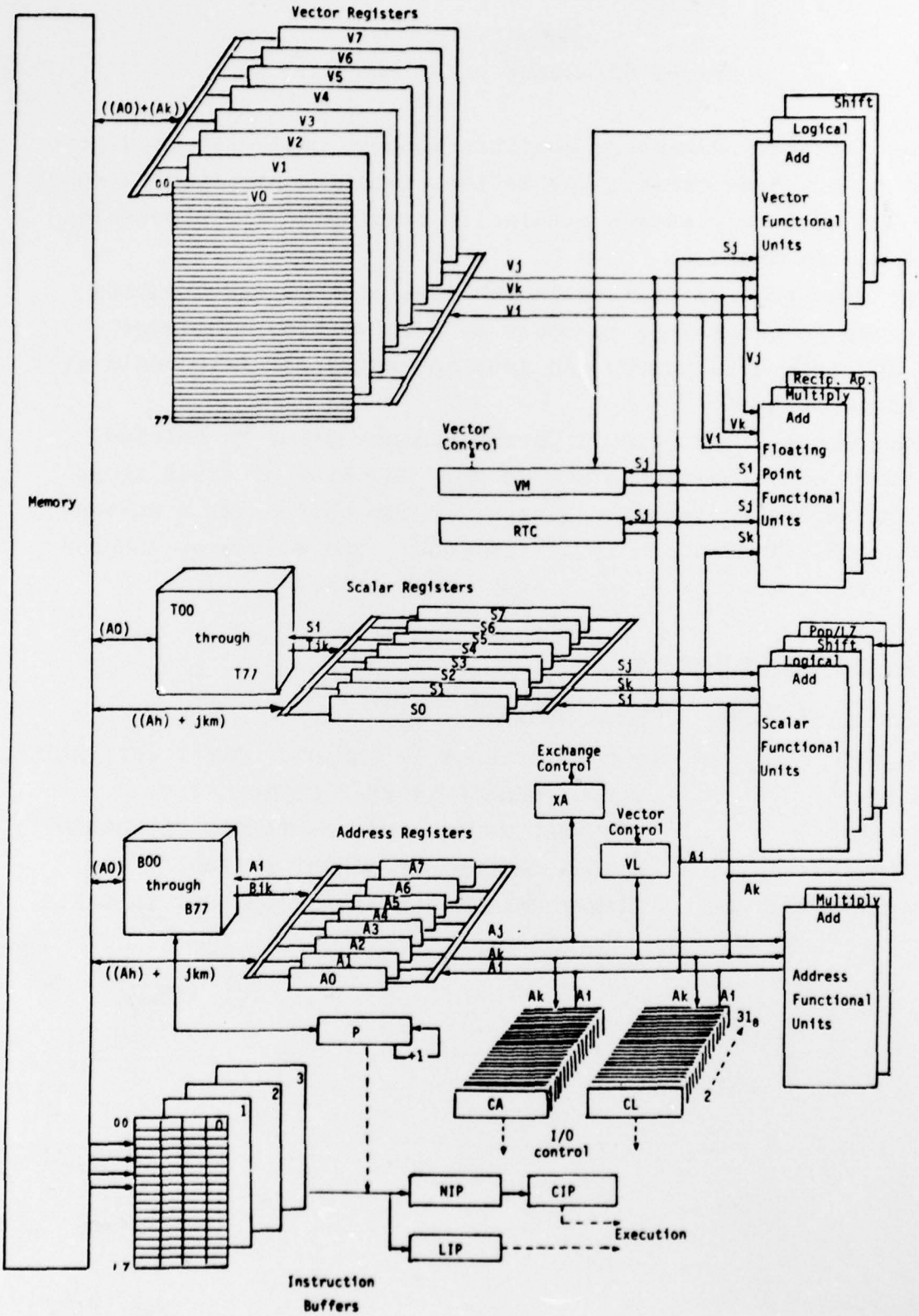


Figure E-1. Computation section

Appendix F.
Cray-1 Simulator Error Stops

This appendix discusses possible (though unlikely) simulator error stops. These error stops reflect internal simulator errors that could adversely affect simulation results if the simulation was allowed to proceed.

The user program that calls the simulator as a subroutine could cause an error stop to occur by over writing simulator tables and data structures. An invalid TABLES.DAT file could also cause an error stop.

Some error stops print an error message prior to halting, other stops only indicate a stop code. The list of error stops below are separated into two groups: those that print a message and those that indicate only a stop code. The simulator subprogram issuing the error stop is also noted below.

Error Stops with Messages

Origin of Stop	Stop Message
TABINI	RECORD OVERRUN IN TABINI. CAN'T INITIALIZE.
TABINI	TABINI UNABLE TO READ DECODE FILE.
DECODE	INTERNAL ERROR. DECODE TABLES CLOBBERED.
SIMBRK	SIMBRK CALLED BUT BRKSET ≤ ZERO.
SIMBRK	SIMBRK CALLED BUT BREAKPOINT NOT IN TABLE.

Error Stops With Stop Codes

Origin of Stop	Stop Code	Comments
MSW	101	Invalid bit code in MSW.
MSW	102	Invalid argument to MSW.
SIMCON	103	Unimplemented action code used.
SIMCON	104	" " " " " .
QPROC	105	Invalid queue action code.
QWRITE	106	Queue space exhausted.
QWRITE	107	Invalid queue pointer.
SETMSK	108	Invalid bit code in SETMSK.
BLDMSK	109	Invalid hold issue code.
DECODE	113	Bad instruction format code.
SIMCON	114	Invalid action code used.
RESERV	115	Invalid reservation code.
ACTION	116	Action held and Queue empty
QWRITE	118	Invalid clock period argument.
RESG00	200	Invalid G-field dispatch code.
RESG01	201	" " " " .
RESG02	202	" " " " .
RESG03	203	" " " " .
RESG04	204	" " " " .
RESG05	205	" " " " .
RESG06	206	" " " " .
RESG07	207	" " " " .
RESG14	214	" " " " .
RESG15	215	" " " " .
RESG16	216	" " " " .
RESG17	217	" " " " .
TRACK	300	Invalid track command code.
ENTRAP	400	Floating point interrupt process failure.
ENTRAP	401	" " " " " .
ENATTN	402	Attention process failure.
ENATTN	403	" " "
RESG07	1071	Invalid J-field dispatch code.

Appendix G.

Program Availability Information

Name:	Cray-1 Simulator
Language:	IBM 360/370 Fortran-IV IBM 360/370 Assembly language Major revision required to run on a non-IBM 360/370 series machine. (32 bit word, byte addressability)
Storage:	270,000 bytes with a 4,096 word Cray-1 memory.
Operating System Requirements:	The simulator runs on the Michigan Terminal System (MTS). Changes will be necessary to interface with other operating systems and other Fortran I/O systems.
Other Requirements:	none
Availability:	Source language programs available from Professor D.A. Calahan on 9-track, 800 BPI magnetic tape. The program is roughly 11,000 records long.

Professor D.A. Calahan
University of Michigan
Dept. of Electrical and Computer Engineering
Ann Arbor, MI 48109
(313) 763-0036

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

Appendix H.

Sample Simulator Exit Dispatcher

```
      SUBROUTINE CRAYEX(IJK, A$R, S$R, V$R, VL, EXSW)
      IMPLICIT INTEGER(A-Z)
      LOGICAL EXSW, NOANS/.TRUE./
      INTEGER A$R(8), RTC(2)
      REAL*8 S$R(8), V$R(64,8), DS
      EQUIVALENCE(RTC(1),DS)
C
C
C ... THIS EXIT PROCESSOR IS USED BY A FULL MATRIX LU FACTORIZATION
C     PROGRAM.  TWO EXIT FUNCTIONS ARE PROVIDED:
C
C     EX  1 - INITIALIZES THE SQUARE MATRIX IN CRAY-1 MEMORY.
C             REGISTER A1 POINTS TO THE MATRIX.
C             REGISTER A2 CONTAINS THE MATRIX SIZE.
C
C     EX  2 - PRINTS OUT THE RUN TIME AND THE MATRIX SIZE.
C             REGISTER A7 CONTAINS THE MATRIX SIZE.
C             REGISTER S7 CONTAINS THE REAL TIME CLOCK VALUE.
C
C             OPTIONALLY, IF THE LOGICAL VARIABLE 'NOANS' IS .FALSE.,
C             THEN THE "EX  2" WILL ALSO PRINT THE MATRIX SOLUTION.
C
C
C   COMMON BLOCK FOR CRAY-1 MEMORY.
C
      DOUBLE PRECISION MEM
      COMMON /MSIZE/  MEMSIZ
      INTEGER*2                                   HMEM(1)
      INTEGER                        IMEM(2,4096)
      COMMON /MEMORY/ MEM(4096)
      EQUIVALENCE    (MEM(1),   IMEM(1,1),   HMEM(1))
C
C
C
C
C ... DISPATCH ON THE EXIT CODE (IJK)
C
      GO TO(100,200), IJK
      EXSW=.TRUE.
      RETURN
C
C ... CODE=1 INITIALIZE MATRIX  (A1->BASE, A2=SIZE)
100   N=A$R(2+1)
      MADDR=A$R(1+1)
      SMADDR = MADDR
      DO 140 J=1,N
      K=J
      DO 130 I=1,N
      MEM(MADDR+1) = K
      MADDR=MADDR+1
      K=K-1
      IF(K.LT.1) K=N
130   CONTINUE
140   CONTINUE
      RETURN
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
C
C
C ... CODE=2 PRINT THE TIME.  (A7 = MATRIX SIZE, S7 = RUN TIME)
200   DS = SSR(7+1)
      WRITE(6,1000) ASR(7+1), RTC(2)
1000  FORMAT(' SIZE=',I5,'  RTC=',I7)
C     RXSW = .TRUE.
      IF(NOANS) RETURN
C
      DO 250 I=1,N
250   WRITE(6,1100) (MEM( SMADDR+(J-1)*N+I ),J=1,N)
1100  FORMAT(1X,10F7.3)
C
      RETURN
C
C
      END
```

Appendix 1.

Sample Simulator Calling Program

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
      IMPLICIT INTEGER(A-Z)
C
C
C ... THIS MAIN PROGRAM CALLS THE CRAY-1 SIMULATOR AS A SUBROUTINE
C     TO SOLVE PARALLEL SYSTEMS OF TRI-DIAGONAL EQUATIONS.  UP TO 64
C     PARALLEL SYSTEMS MAY BE SOLVED.
C
C     THIS PROGRAM PERFORMS THE FOLLOWING FUNCTIONS:
C
C        1. READS TWO INPUT PARAMETERS:
C             NSYS - THE NUMBER OF PARALLEL SYSTEMS TO SOLVE.
C             NEQS - THE NUMBER OF EQUATIONS IN EACH SYSTEM.
C
C        2. ALLOCATES THE CRAY-1 MEMORY FOR THE THREE DIAGONALS AND THE
C           RIGHT HAND SIDE.
C
C        3. INITIALIZES THE SYSTEMS.
C
C        4. LOADS THE CRAY-1 TRI-DIAGONAL LU DECOMPOSITION ROUTINE (TRIDEC)
C           AND INITIALIZES THE CALLING PARAMETERS IN AN ADDRESS LIST
C           IMMEDIATELY PRECEDING THE LOADED PROGRAM.
C
C        5. GIVES CONTROL TO THE USER VIA THE SIMULATOR COMMAND LANGUAGE,
C           ALLOWING THE USER TO RUN THE PROGRAM, SET BREAK POINTS, ETC.
C
C        6. UPON RETURN FROM THE SIMULATOR, PRINT OUT THE SYSTEM AND
C           CALCULATE THE MFLOPS.
C
C        7. LOADS THE BACK-SUBSTITUTION CRAY-1 PROGRAM (TRISLV) AND INITIALIZES
C           ITS CALLING PARAMETERS.
C
C        8. AGAIN GIVES CONTROL TO THE USER TO RUN THE PROGRAM.
C
C        9. UPON RETURN FROM THE SIMULATOR, PRINT OUT THE SYSTEM AND
C           CALCULATE THE MFLOPS.
C
C
C        THIS PROGRAM TAKES THE PLACE OF THE SIMULATOR'S MAIN PROGRAM SINCE
C        IT IS LOADED FIRST.  IT ALSO EXTENDS THE SIMULATOR MEMORY TO 8192
C        WORDS.
C
C
      COMMON /PARMS/ NSYS, NEQS, ABASE, BBASE, CBASE, YBASE
C
C
C ... THE FOLLOWING IS AN EXTENSION OF THE CRAY-1 SIMULATED MEMORY.
C
      DOUBLE PRECISION MEM
      COMMON /MEMORY/ MEM(8192)
      COMMON /MSIZE/  MEMSIZ
      INTEGER*2 HMEM(32768)
      INTEGER   IMEM(2,8192)
      EQUIVALENCE (MEM(1),IMEM(1,1),HMEM(1))
C
C ... TELL THE SIMULATOR THE NEW SIZE OF CRAY-1 MEMORY.
      MEMSIZ = 8192
      CALL CRAY1('INIT!', .TRUE.)
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
C
C
C
C ... II - THE I DIRECTION INCREMENT.
C           THE DISTANCE BETWEEN DIAGONAL ELEMENTS.
C ... IJ - THE J DIRECTION INCREMENT.
C           THE DISTANCE BETWEEN PARALLEL ELEMENTS.
C
C
      II = 1
      READ(15,1000) NSYS, NEQS
1000  FORMAT(I5/I5)
      IF(NSYS .GT. 64) GO TO 910
C
      IJ = NEQS
C
C ... SET ARRAY BASES.
      CBASE = 300
      ABASE = CBASE + NSYS*NEQS
      BBASE = ABASE + NSYS*NEQS
      YBASE = BBASE + NSYS*NEQS
C
      IF(YBASE + N*K .GT. MEMSIZ) GO TO 900
C
C ... INITIALIZE TRIDEC CRAY MEMORY WITH THE TRI-DIAGONAL DATA.
C
C ... LOOP THRU THE ELEMENTS OF A SYSTEM TO INITIALIZE.
      DO 10 I = 1,NEQS
C
C ...   LOOP THRU ALL PARALLEL SYSTEMS.
        DO 10 J = 1,NSYS
        MEM(CBASE+(J-1)*IJ + (I-1)) = 0.1D0
        MEM(ABASE+(J-1)*IJ + (I-1)) = 1.0D0
        MEM(BBASE+(J-1)*IJ + (I-1)) = I/10.0D0
        MEM(YBASE+(J-1)*IJ + (I-1)) = I*1.0D0
10      CONTINUE
C
C ... CLEAR OUT THE TOP OF C AND THE BOTTOM OF B.
      DO 20 J = 1,NSYS
        MEM(CBASE+(J-1)*IJ + (1-1)) = 0.0D0
        MEM(BBASE+(J-1)*IJ + (NEQS-1)) = 0.0D0
20    CONTINUE
C
      WRITE(14,1200)
1200  FORMAT('1CRAY-1 TRI-DIAGONAL SOLVER')
C
      CALL CHECK(0, .FALSE.)
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
C
C ... SET UP THE ARGUMENT CALL BLOCK WITH POINTERS TO
C     THE ARGUMENTS.
C
C    LOC 100 = NSYS,  LOC 101 = NPOS,  LOC 102 = II,  LOC 103 = IJ,
C    LOC 104 = CLOCK.  (IMEM(1) = CRAY-1 ADDRESS ZERO.)
C
      IMEM(2,9+1)  = 68
      IMEM(2,10+1) = 67
      IMEM(2,11+1) = 66
      IMEM(2,12+1) = BBASE-1
      IMEM(2,13+1) = ABASE-1
      IMEM(2,14+1) = CBASE
      IMEM(2,15+1) = 65
      IMEM(2,16+1) = 64
C
C ... SET UP ARGUMENT LOCATIONS.
      IMEM(2,68+1) = 0
      IMEM(2,67+1) = IJ
      IMEM(2,66+1) = II
      IMEM(2,65+1) = NPOS
      IMEM(2,64+1) = NSYS
C
C
C ... LOAD TRIDEC AND GIVE SIMULATOR CONTROL TO THE USER.
      CALL CRAY1('COM AFTER TRIDEC LOADS, RUN 21A TO START.!', .FALSE.)
      CALL CRAY1('LOAD SGTG:TRIDEC;USE *MSOURCE*!', .FALSE.)
C
C
      NOPS = NSYS * (1 + (NPOS-1)*4)
      CALL CHECK(NOPS, .TRUE.)
C
C ... INITIALIZE TRISLV'S ARGUMENT BLOCK WITH ITS POINTERS.
C
      IMEM(2,10+1) = 68
      IMEM(2,11+1) = YBASE-1
      IMEM(2,12+1) = 67
      IMEM(2,13+1) = 66
      IMEM(2,14+1) = BBASE - 1
      IMEM(2,15+1) = ABASE-1
      IMEM(2,16+1) = CBASE
      IMEM(2,17+1) = 65
      IMEM(2,18+1) = 64
C
C
C ... LOAD TRISLV AND GIVE SIMULATOR CONTROL TO USER.
      CALL CRAY1('COM AFTER TRISLV LOADS, RUN 23A TO START.!', .FALSE.)
      CALL CRAY1('LOAD SGTG:TRISLV;USE *MSOURCE*!', .FALSE.)
C
      NOPS = NSYS * ((NPOS-1)*2 + NPOS*3)
      CALL CHECK(NOPS, .TRUE.)
C
C
      STOP
```

```
C
C ... NOT ENOUGH MEMORY FOR THE PROBLEM SIZE.
900   MAXPRB = (MEMSIZ - CBASE) /4
      WRITE(6,9000) MAXPRB
9000  FORMAT('0CRAY-1 MEMORY TOO SMALL FOR THIS PROBLEM SIZE.'/
     +       ' THE LARGEST PRODUCT OF NSYS*NEQS MUST BE < ',I5)
      STOP
C
C ... TOO MANY PARALLEL SYSTEMS.
910   WRITE(6,9010) NSYS
9010  FORMAT('0THE NUMBER OF PARALLEL SYSTEMS MAY NOT EXCEED 64.'/
     +       ' ',I5,' WAS SPECIFIED.')
      STOP
C
C
      END
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
      SUBROUTINE CHECK(NOPS, PTIME)
      IMPLICIT INTEGER(A-Z)
      COMMON /PARMS/ NSYS, NEQS, ABASE, BBASE, CBASE, YBASE
C
C   COMMON BLOCK FOR CRAY-1 MEMORY.
C
      DOUBLE PRECISION MEM
      COMMON /MEMORY/ MEM(8192)
      COMMON /MSIZE/  MEMSIZ
      INTEGER*2 HMEM(32768)
      INTEGER   IMEM(2,8192)
      EQUIVALENCE (MEM(1),IMEM(1,1),HMEM(1))
C
      REAL MFLOPS
      LOGICAL PTIME
C
C
C ... CALC MFLOPS
C
      RTC = IMEM(2,68+1)
      MFLOPS = 0.0
      IF(RTC.NE.0) MFLOPS = (NOPS * 80.0)/RTC
C
C ... PRINT THE RESULTS
      WRITE(14,4000)
      DO 40 I = 1,NEQS
40    WRITE(14,5000)  I,MEM(CBASE+I-1), I,MEM(ABASE+I-1),
     +                I,MEM(BBASE+I-1), I,MEM(YBASE+I-1)
      IF(PTIME) WRITE(14,6000) RTC, NSYS, NEQS, MFLOPS
C
4000  FORMAT('-')
C
5000  FORMAT( '  C(',I2,')=',E13.4, '  A(',I2,')=',E13.4,
     +        '  B(',I2,')=',E13.4, '  Y(',I2,')=',E13.4)
C
6000  FORMAT(' RTC =',I7 ,'    * SYSTEMS = ',I3, '   SIZE OF SYSTEM =',
     +       I3,'   MFLOPS =',F12.3,/' ')
C
      RETURN
      END
```

Appendix J.

Glossary of MTS Terms

For those readers unfamilier with MTS, this glossary defines the MTS terms used in this report.

Attention Interrupt -

The attention interrupt is a means of allowing the MTS time-sharing user to regain control over an errant job. The user at a terminal may signal an attention by hitting the terminal break key or typing a control-E. If the program being run does not intercept the attention, MTS will field it and stop the program.

FDUB -

A file or device usage block (FDUB) is a data structure used by MTS to control I/O to files and devices. A program may request MTS to create a FDUB for a file or a device that is to be accessed. Upon creation of the FDUB, MTS returns to the program with a pointer to the FDUB. The program may then use this FDUB pointer as an argument to the MTS subroutines READ and WRITE which will control the I/O to the file or device.

The simulator's command stream input stack is simply a stack of FDUB pointers to all the files currently open. Within each FDUB, MTS remembers the line number of the next record to be read.

Keyboard Attention -

See Attention Interrupt.

MTS -

MTS, the Michigan Terminal System, is a terminal-oriented time-sharing operating system that offers both batch and terminal facilities. Currently, MTS is run on an Amdahl 470/V6 computer at the University of Michigan.

MSINK -

MSINK is a pseudo-device name used by MTS to refer to the master sink (master output) device for the job. In batch mode, the master output device is the line printer. In terminal mode, the master output device is the terminal printer.

MSOURCE -

MSOURCE is a pseudo-device name used by MTS to refer to the master source (master input) device for the job. In batch mode, the master input device is the card stream. In terminal mode, the master input device is the terminal keyboard.

PRINT -

PRINT is the pseudo-device name used by MTS to refer to the line printer. This allows a terminal user to direct output to the line printer by specifying *PRINT* as the output device. MTS assigns the terminal user a receipt number used to pick up the output.

SOURCE -

SOURCE is a psuedo-device name used by MTS to refer to the current input device. MTS reads its commands from this pseudo-device. When a terminal or batch job begins, *SOURCE* is the same as *MSOURCE*. But, unlike *MSOURCE*, whose definition is fixed, *SOURCE* can refer to another file or device. The definition of *SOURCE* is under user control via the MTS command "$SOURCE fdname", where fdname is the file of device name which the user wishes to be defined as *SOURCE*. Therefore, any program, or MTS, which reads from *SOURCE* will read from the file or device specified in the $SOURCE command.

SINK -

SINK fills the same role for the current output device that *SOURCE* does for the current input device. The user may change *SINK* through the MTS command "$SINK fdname". When a terminal or batch job begins, *SINK* is defined as *MSINK*.

Appendix K.
Load Module Formats

1. Absolute Modules

An absolute load module is composed of a header record and one or more parcel records. The header record is composed of three free formatted fields: a starting parcel address, a module length and an optional title as shown below.

p-addr length [title-string]

Each field is separated by blanks. The p-addr field contains a modified octal (octal word address followed by an A,B,C or D parcel code) parcel address where the first parcel is stored. The length field contains the octal number of parcels following on subsequent parcel records. The title-string field contains the title that is placed in the title field of the CPACT report (also see the SET TITLE = title-string command). The title field begins with the first character after the blank which terminates the length field and may contain embedded blanks. Only the first 35 characters are retained. The LOAD command will echo the header record information to the output device.

The parcel record portion of the absolute load module contains one or more free formatted records. One or more parcels may be provided on each record in octal or modified octal format and must be separated by at least one space. A colon may be placed anywhere on a parcel record which will stop the scan of the record at that point. This allows comments to be placed to the right of the colon. Blank records in the parcel record portion are ignored. The number of parcels in the parcel record portion of the load module must correspond to the octal length specified on the header record.

All records in an absolute load module must be less than 81 characters. Since the load modules are represented in character format it is a simple matter to patch a module with the editor. If you add or delete parcels, you must update the octal length appropriately.

Example:

```
21A  3  A TITLE
:  THIS IS AN EXAMPLE OF A 3 PARCEL
:  ABSOLUTE LOAD MODULE
022133  032211  :  A1  33  ;  A2  A1*A1
004000  :  EX  0
$ENDFILE
```

2. Relocatable Modules

Relocatable modules consist of seven types of binary records. An IDEN record, one or more TXT records, zero or more RLD, EXT, ENTR, and SYM records, and an END record.

An IDEN record identifies the name of the module. The record consists of the characters IDEN, followed by 4 spaces, followed by the 8 character name of the module.

A TXT record contains the actual object code to be loaded. It consists of the letters TXT, followed by one space, followed by a four byte binary address of this portion of the module (relative to the top of the module), followed by a four byte binary length. The actual text to be loaded is on the following card.

An RLD record identifies the locations in the module which must be relocated. It consists of the letters RLD, followed by one space, followed by one or more 8 byte fields. The first 4 bytes of the field contain the binary address (relative to the top of the module) of the text to be relocated. The second 4 bytes contain a number describing the type of relocation to be performed. See RLD & EXT types, below.

An EXT record identifies the locations in the module which refer to external locations. It consists of the letters EXT, followed by 5 spaces, followed by one or more 16 byte fields. The first 8 bytes of the field contains the 8 character name of the external location referenced. The next four bytes contain the binary address (relative to the top of the module) of the text referencing the external. The last 4 bytes contain a number describing the type of reference. See RLD & EXT types, below.

An ENTR record identifies entry points in the module. It consists of the letters ENTR, collowed by 4 spaces, followed by one or more 12

byte fields. The first 8 bytes of the field contain the name of the entry point, and the last 4 bytes contain the address (relative to the top of the module) of the entry point.

A SYM record contains definitions of all symbols in the module which may be referenced by the simulator command language. It consists of the letters SYM, followed by 5 spaces, followed by one or more 16 byte fields. The first 8 bytes contain the name of the symbol. The next 4 bytes contain the value of the symbol. The last 4 bytes contain a number identifying the type of the symbol. See SYM types, below.

An END record terminates the module. It consists of the letters END, followed by 1 space. If a START pseudo op was contained in the program, the next 4 bytes contain an address (relative to the top of the module) of the starting location.

RLD and EXT types:
1: Two parcel, parcel address
2: Two parcel, word address
3: Four parcel, parcel address
4: Four parcel, word address

SYM types:
1: Parcel
2: Value
3: Word