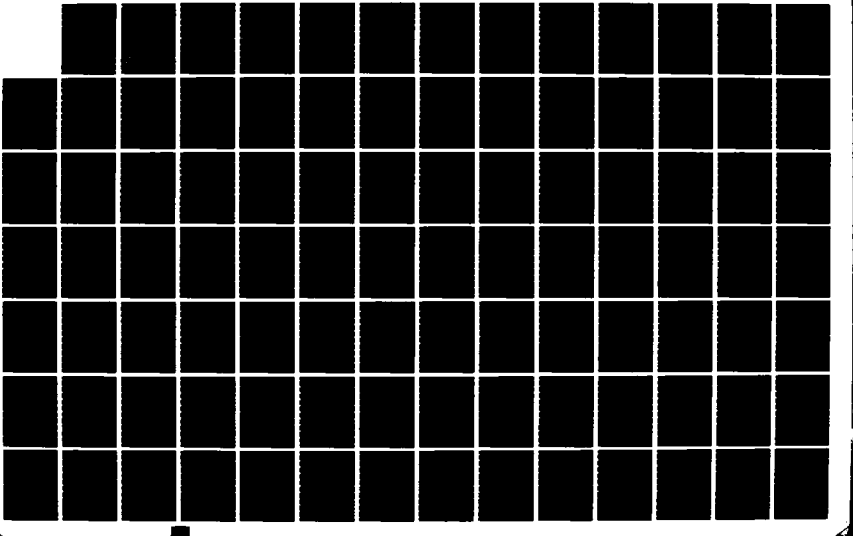
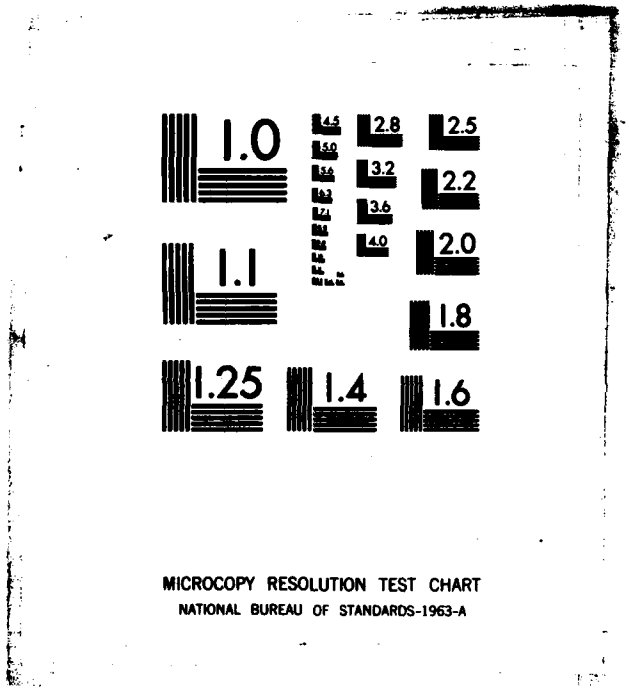
AD-A136 555 A CRAY-CLASS MULTIPROCESSORSSIMULATOR(U) MICHIGAN UNIV 1/2
ANN ARBOR SUPECCOMPUTERAALGORITHM RESEARCH LAB
P M SUMMERS ET AL. 01 SEP 83 SARL-1 AFOSR-RR-83-1246
UNCLASSIFIED AFOSR-80-0158 F/G 9/2 NL





Report SARL #1

A CRAY-class Multiprocessor Simulator

P.M. SUMMERS
D.A. ORBITS

September 1, 1983

Sponsored by the
Directorate of Mathematical & Information Sciences
Air Force Office of Scientific Research

Supercomputer Algorithm Research Laboratory
Department of Electrical & Computer Engineering

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 83-1246	2. GOVT ACCESSION NO. AD-A136 555	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A CRAY-CLASS MULTIPROCESSOR SIMULATOR		5. TYPE OF REPORT & PERIOD COVERED TECHNICAL 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) P.M. Summers and D.A. Orbits		8. CONTRACT OR GRANT NUMBER(s) AFOSR-80-0158
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Electrical and Computer Engineering University of Michigan Ann Arbor MI 48109		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS PE61102F; 2304/A3
11. CONTROLLING OFFICE NAME AND ADDRESS Mathematical & Information Sciences Directorate Air Force Office of Scientific Research/NM Bolling AFB DC 20332		12. REPORT DATE 1 SEP 83 13. NUMBER OF PAGES 128
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

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)

Supercomputers; parallel processors; vector processors; simulation.

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

A logical-timing instruction-level simulator is described for a hypothetical multiprocessor consisting of CRAY-1's connected to a common memory. It is useful for gaining insight into the design of multiprocessor algorithms and for developing high performance algorithms for CRAY processors with instruction sets similar to the CRAY-1.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

A Cray Class Multiprocessor Simulator

Paul M. Summers
D. A. Orbits

SuperComputer Algorithm Research Laboratory

University of Michigan

Ann Arbor, Michigan 48109

September 1, 1983

Grant # - AFOSA-80-0158

Abstract

A logic-timing simulator is described for a hypothetical multiprocessor consisting of CRAY-1's connected to a common memory. This simulator is written in Fortran-IV and IBM assembly to execute on an Amdahl 5860 machine, operating under the Michigan Terminal System.

The simulator provides extensive reporting of individual CRAY-1 processor resource usage and resource conflicts and interprocessor communication. By calling the simulator as a subroutine the user may flexibly use program simulation within a larger problem environment. Extensive interactive debugging features make the CRAY multiprocessor simulator a useful tool for (1) gaining general insight into the design of multiprocessor algorithms, and (2) the development of assembly language programs for CRAY processors with instruction sets similar to the CRAY-1.

Table of Contents Page

Preface....ii
1. Introduction....1
2. Simulator Features....3
2.1 Command Language....4
2.2 Exceptional Conditions....11
2.3 Subroutine Interface....15
2.4 CRAYEX Exit Dispatcher....19
2.5 Report Generation....21
2.6 Inconsistencies with the Cray-1....56
3. Simulator Command Descriptions....57
4. Simulation Costs....97
5. Bibliography....99

Appendices

A. Summary of Cray-1 Timing Information....100
B. Simulator I/O Device Usage....105
C. Simulator Common Block Usage....106
D. Establishing the Simulator on MTS....107
E. CRAY-M Shared Registers and Semaphores....108
F. Simulator Error Messages and Error Stops....110
G. Program Availability Information....112
H. Sample Simulator Exit Dispatcher....113
I. Sample Simulator Calling Program....115
J. Load Module Formats....119
K. Task Definition File Description....120
L. Example Use of Cross-Assembler and Simulator....122

Preface

The simulator described in this report was developed to support general vector multiprocessor algorithm studies. It was felt to be of sufficient general interest and utility that this documentation was prepared.

The simulator accepts machine code from a cross assembler developed at the University of Michigan and described in SEL Report #120 and in the Appendix of this report.

Both the cross-assembler and the simulator will be available from Professor D. A. Calahan in January, 1984.

1. Introduction

The original University of Michigan Cray-1 (uniprocessor) simulator was written during 1977-78 by D. A. Orbits. The decision to build a simulator was motivated by the following considerations:

(1) At the time, 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.

(2) Even with access to a Cray-1, it was often quite difficult to analyze algorithm performance. There was no hardware instrumentation on a Cray computer to permit a study of CPU resource usage and conflict. The Cray-1 simulator provided a detailed report of CPU activity.

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

(4) When debugging programs, it was 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. (Note: The CTSS operating system now provides many of these capabilities.)

(5) With simulation it was possible to study the impact of architectural modifications on algorithm performance.

A somewhat similar situation exists with respect to the Cray XMP and other presently unannounced Cray multiprocessors. Availability is currently restricted. Although the significance of assembly language (CAL) coding may be reduced in future machines, there is a new requirement to study the organization and efficiency of various tasking strategies on kernels, scientific libraries, and entire application programs.

The simulator described in this report is intended to support such study. It contains two major extensions of the CRAY-1 simulator

(a) A number (p, = 4 but alterable) of CRAY-1's are connected to the same common memory. Each processor has the instruction set and timings of the CRAY-1. This is, of course, a hypo-

thetical or paper machine. Intraprocessor but not interprocessor bank conflicts are modeled.

(b) Hardware semaphores and shared registers have been added to the CRAY-1 architecture (see Appendix L), and assembly instructions are included similar in format to those of the Cray XMP, to assist in program development for this machine. However, the timing of these instruction executions is different from the Cray XMP, and may be changed as we feel appropriate. Thus, the timings produced by this simulator are advisory, visa vis the precise timings of the parent Cray-1 simulator.

In this report, the designation Cray-1 will be used to denote one of the processors or its instruction set; the term Cray-M will denote the entire simulated multiprocessor.

In summary, this software can, at a minimum, yield insight into the interplay of hardware and algorithms by direct control from CAL of the hardware multitasking facilities. Beyond this, it may be that certain high-performance library routines and other algorithms requiring complicated tasking and sub-tasking strategies can be best implemented with the simulator, analagously to the CAL HYPAC linear algebra library developed by the Cray-1 simulator.

2. Simulator Features

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 is 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.

(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, STAT, TACT, and TRACE 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-M 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 three sub-sections:

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

2.1.1 Command Language Input Control

Upon initiation, 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

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 USE *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 I/O unit by using the SET command to switch the output device. For example, SET OUTPUT = -F1. would route the output to file "-F1".

Error messages are output on a different unit number and always go to *MSINK*. 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 Running Programs on the Cray-M 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-M 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 I.

When designing a Cray-M program to be simulated, consideration must be given first to the nature of the algorithm under study. If the program requires some initialization which will not 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-M memory. In fact, the user may increase the size of the simulated Cray-M memory beyond the 4096 IBM double-words that are presently allocated.

2.1.4 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.

If the algorithm under study requires the use of intrinsic functions, such as SQRT, SIN, COS, etc, which would be supplied by some Cray-M subroutine library, the user may provide these functions through the use of Cray-M simulator EXIT instruction dispatcher. The 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-M 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 a Cray 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 Cray arithmetic, further raising the simulation cost. As a consequence of using host machine (IBM 370) arithmetic, 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.

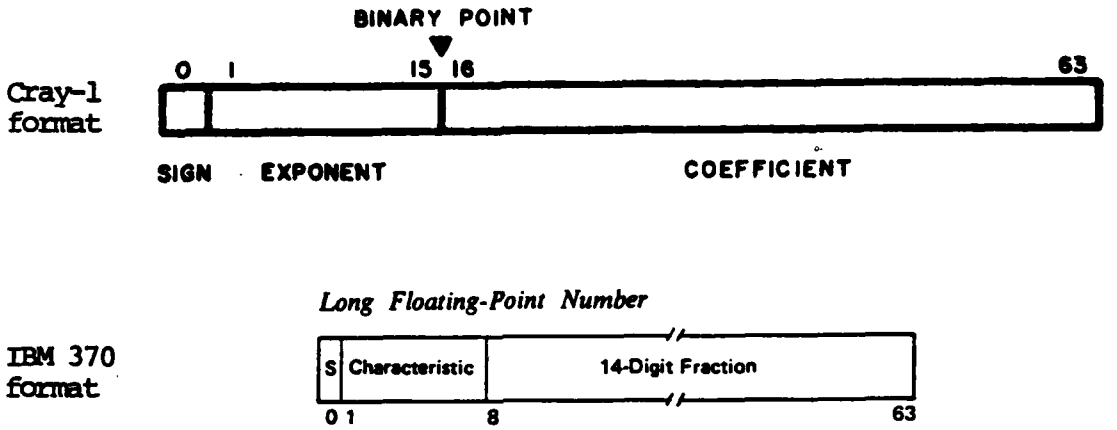


Figure 2.1.1 - Cray-1 vs. IBM 370 Floating point data formats

The simulation of the instruction computation is done in its 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-M 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 the current program counter locations and cause control to return to the command language after 2500 Cray-1 instructions have been issued in at least one processor (unless an EXIT instruction or error condition occurred).

```
RUN      #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
```

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

See Section 4 for command descriptions of all simulator commands.

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

Level 1 - Result computation only, which allows debugging but eliminates the cost associated with timing the Cray-1 instructions.

Level 2 - Timing enabled, allowing the timing of the simulated program at a cost of about 5 times the level one cost, per processor.

Level 3 - CPACT (clock period activity report) enabled, increases the cost to about 20 times the level one cost, per processor.

Section five treats the cost issue further.

2.2 Exceptional Conditions

While executing a Cray-M program, the simulator may encounter any of several exceptional conditions which will halt the simulation. The four 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) Invalid instruction executed

An occurrence 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-M 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-M 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-M 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 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.

2.2.5 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 suppress if the floating point interrupt bit in the Cray-1 mode register clear. When the simulator starts up, this mode register bit is thereby enabling all three types of floating point interrupts. setting of this mode register bit may be controlled by the user 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 inst simulated. This means that if a vector instruction causes 20 exp overflows, only one will be detected. After the instruction has finished executing the simulator will announce the floating poin 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.

2.2.6 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-M 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-M, 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-M 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 #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 #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-M 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-M 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-M memory. MEMSIZ is used to perform bounds checking on branches and memory references made by the simulated Cray-M instructions. When the simulator is called for the first time some once-only initialization is done which includes zeroing all of Cray-M memory (MEM). Therefore, MEMSIZ must be initialized properly before the first call to CRAY1. Further since the once-only initialization will zero Cray-M 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-M 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-M 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-M memory, the user need only read and write data to the MEM array. However, because the Cray-M 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-M memory address and adding one to it. For example, Cray-M 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;USE *MSOURCE*!',.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-M 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-M program. This transfer of control from the Cray-M 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 of the CPU that executed the EX instruction to CRAYEX. This allows arguments to be provided and results returned through the A-registers. Cray-1 register A0 corresponds to AREG(1).

SREG - This parameter is an eight element double precision array used to pass the S-register contents of the CPU that executed the EX instruction to CRAYEX. This allows arguments to be provided and results returned through the S-registers. Cray-1 register S0 corresponds to SREG(1).

VREG - This parameter is a double precision array, dimensioned as (64,8), used to pass the vector register contents of the CPU that executed the EX instruction to CRAYEX. Arguments may be provided and results returned through the vector registers. Cray-1 vector register V0 corresponds to VREG (-,1).

VL - This parameter is an integer which contains the value of the vector length register of the CPU that executed the EX instruction. On entry to CRAYEX, VL will always be between 1 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.

In addition to the CRAYEX calling parameters, the CRAYEX program may access Cray-M 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-M 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-M simulator produces five kinds of report outputs, STAT, CPACT, TRACE, TACT and TACT STAT. The STAT report is a summary report of the program's use of Cray-M resources, i.e., across all Cray-1 processors. The CPACT report is a detailed report of individual Cray-1 resource usage at each clock period of the program's

execution. The TRACE report is a flowtrace which, for each executed instruction, displays for each CRAY-1 the instruction mnemonic, the instruction address, and the contents of the storage locations that the instruction affects. The TACT report shows which task each CPU is executing at constant clock period intervals, analogous to the CPACT clock-level report for each processor. The TACT STAT report is a summary report of all task activity, analogous to the STAT clock-level summary. For p processors, a total of 2p + 3 reports can be generated for each run.

2.5.1 STAT report

The STAT report summarizes clock-level activity across all processors and 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 a Cray vector unit resources. Figure 2.5.1, below, shows the Vector Usage Counts table.

U OF M CRAY-M SIMULATOR (UM138)
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-M 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 its 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. To receive a floating point result count report, the FULL option must be specified on the STAT command.

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 paths of the Cray-M. To aid in identifying the various data paths for which traffic information is provided, the simulator prints a block diagram of a Cray-1 uni-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-M 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-M 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 dotte

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

The right hand column of the figure represents Cray-M 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

2.5.1.1 STAT Example

To illustrate the information provided by the STAT command, one example is presented. The code in this example is one which multiplies four pairs of matrices together. After running the program with timing turned on (SET TIM=ON), the STAT FULL command is given, producing a STAT Report containing all three sections (Vector Usage, Floating-point Result and Data Traffic).

U OF M CRAY-M SIMULATOR (EP011)

VECTOR USAGE COUNTS

CUM. TIMING	FP ADD	FP MUL	FP DIV	LOG.	SHIFT	I. ADD	V-LOAD	V-STOR
TIME BUSY (CP)	136	68	0	136	0	0	136	69
% TIME BUSY	34.78%	17.39%	0.0 %	34.78%	0.0 %	0.0 %	34.78%	17.65%
NO. RESULTS	128	64	0	128	0	0	128	64
NO. VECTORS	2	1	0	2	0	0	2	1
AVERAGE VL	64.00	64.00	0.0	64.00	0.0	0.0	64.00	64.00

RUN TIME (CP)	:	391
MFLOPS	:	39.28
COMPOSITE AVL	:	64.00
CONCURRENCY	:	1.39
MIPS	:	16.16

FLOATING POINT RESULT COUNTS

	ADDITION	MULTIPLICATION	RECIPROCAL	TOTAL
VECTOR (%)	128 (66.7)	64 (33.3)	0 (0.0)	192 (100.0)
SCALAR (%)	0 (0.0)	0 (0.0)	0 (0.0)	0 (0.0)
TOTAL (%)	128 (66.7)	64 (33.3)	0 (0.0)	192 (100.0)

Figure 2.5.4. STAT Example

```
                    DATA  TRAFFIC  COUNTS

                                  V U    VO      V       VMO      M
VO     :        512               E N  <======  E R  <======    E
VMO    :        128               C I           C E             M
                                  T T           T G             O
VR     :        320               O S  ======>  O .  ======>    R
VMR    :         64               R      VR     R      VMR      Y


SXV    :          9                                    SXV


                                  S U    SO      S       SMO      M
SO     :          0               C N  <======  C R  <======    E
SMO    :          0               A I           A E             M
                                  L T           L G             O
SR     :          0               A S  ======>  A .  ======>    R
SMR    :          0               R      SR     R      SMR      Y


SXT    :          0                                    SXT


BTO    :          0                                    T       BTO      M
                                                          R  <======    E
SXA    :          0                          SXA          E             M
                                                          G             O
                                                          B  ======>    R
BTR    :          0                                       BTR      Y


AXB    :         17                                    AXB


                                  A      AO      A       AMO      M
AO     :         52               D U  <======  D R  <======    E
AMO    :          2               D N           D E             M
                                  R I           R G             O
AR     :         26               E T  ======>  E .  ======>    R
AMR    :          0               S S     AR    S       AMR      Y
                                  S             S


MISC.      :        13
BRANCHES   :         4
FETCHES    :         3
ISSUES     :        79

PERCENT OF VECTOR OPERANDS SUPPLIED BY CACHE                    -  75.00%
PERCENT OF TOTAL VECTOR TRAFFIC SUPPLIED BY CACHE               -  76.92%
PERCENT VECTOR RESULTS OF TOTAL RESULTS                         -  92.49%
PERCENT VECTOR MEMORY TRAFFIC OF TOTAL MEMORY TRAFFIC           -  79.34%
RATIO OF COMPUTATION TRAFFIC TO MEMORY TRAFFIC                  -   4.69
RATIO OF VECTOR MEMORY OPERANDS TO VECTOR MEMORY RESULTS        -   2.00
RATIO OF VECTOR UNIT RESULTS TO VECTOR MEMORY OPERANDS          -   2.50
RATIO OF VECTOR UNIT RESULTS TO VECTOR MEMORY RESULTS           -   5.00
RATIO OF VECTOR UNIT CACHE USE TO VECTOR MEMORY CACHE USE       -   4.33
VECTOR MEMORY RESULT RATE                                       -   0.1637 RESULTS/CP
```

Figure 2.5.4. STAT Example (contd)

2.5.2 CPACT Report

The CPACT report produces a detailed clock period activity reford of a Cray-1 uniprocessor state. This is a 132 column report suitable only for printing on a line printer.* The CPACT report can be enabled or disabled for any or all of the CPU's. 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 3) 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, 0-9) 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.

```
                                                   U OF M  C R A Y -  M  S I M U L A T O R   .(UN188)                         PAGE  13

        T                                        PPP       V. REG    S R R R  MEMORY BANKS    A  A. REG  S  S. REG  V AS        S O DF BB
 ST.    A      INSTRUCTION       P-ADDR   CP    PPPVVV               C K K K                  R          R          M 00        T 7 CP ST
        G                                       **/6>+   01234567    1 A B C 0123456789ABCDEF A 01234567 A 01234567   BB        M 3 GA FX

                                         612|       |          |                    | |        | |        | |        |    4
                                         613|       |          |                    | |        | |        | |        |    4
 IS        <BLANK>                       614|       |          |                    | |        | |        | |        |
 IS        <BLANK>                       615|       |          |                    | |        | |        | |        |
 IS  6     B10 A7               121A     616|       |          |                    | |        | |        | |        |
 IS  7     A7  A0-A0            121B     617|       |          |                    | |       7| |        | |        |
 IS  8     B11 A5               121C     618|       |          |                    |7|       7| |        | |        |
 IS  9     B01 A4               121D     619|       |          |                    | |        | |        | |        |
 IS  A     S1  A7               122A     620|       |          |                    | |        | | A      | |        |
 IS  B     B02 A2               122B     621|       |          |                    | |       |A| A      | |        |
 IS  C     S2 #>72              122C     622|       |          |                    | |       |C|        | |        |
 IS  D     S1  S2&S1            122D     623|       |          |                    | |       |D|        | |        |
 IS  E     A7  S1               123A     624|       |          |                    |E|        | |        | |        |
 IS  F     A7  A7+A0            123B     625|       |          |                    | |       F| |        | |        |
 IS  G     B03 A6               123C     626|       |          |                    |F|       F| |        | |        |
 IS  H     VL  A7               123D     627|       |          |                    | |        | |        | |        |
 IS  I     B04 A7               124A     628|       |          |                    | |        | |        | |        |
 IS  J     B05 A3               124B     629|       |          |                    | |        | |        | |        |
 IS  K     B06 A6               124C     630|       |          |                    | |        | |        | |        |
 IS  L     A5  B11              124D     631|       |          |                    |L|        | |        | |        |
 IS  M     A6  A2+A5            125A     632|       |          |                    | |      M | |        | |        |
 IS  N     B12 A1               125B     633|       |          |                    |M|      M | |        | |        |
 IS  O     S1           0,A2    125C     634|       |          |O                   | |        | | O      | |        |
 IS        <BLANK>                       635|       |          | O                  | |        | | O      | |        |
 IS  P     A2  A2+A0            125A     636|       |          |  O                 | |  P     | | O      | |        |
                                         637|       |          |   O                |P|  P     | | O      | |        |
 IS  Q     A7  B10              126B     638|       |          |              O     |Q|        | | O      | |        |
 IS  R     A5  A7-A5            126C     639|       |          |              O     | |     R  | | O      | |        |
 IS  S     A0  A0+A3            126D     640|       |          |              O     |R|S    R  | | O      | |S       |
                                         641|       |          |              O     |S|S       | | O      | |S       |
 IS  T     V3  ,A0,A0           127A     642|       | T        |                    | |        | | O      | |S       |         T
 IS  U     A6  A0+A1            127B     643|       | T        |                    | |U       | | O      | |U       |         T
                                         644|       | T        |                    |U|U      |O| O      | |U       |         T
                                         645|       | T        |                    | |        | |        | |U       |         T
                                         646|       | T        |                    | |        | |        | |U       |         T
                                         647|       | T        |                    | |        | |        | |        |         T
                                         648|       | T        |                    | |        | |        | |        |         T
                                         649|       | T        |                    | |        | |        | |        |         T
                                         650|       | T        |                    | |        | |        | |        |         T
 IS  V     S2           0,A2    127C     651|       | *        |V                   | |        | |  V     | |        |
 IS        <BLANK>                       652|       | T        | V                  | |        | |  V     | |        |
 IS  W     S0  V0,A0            130A     653|       | T        |  V                 | |        | |W V     | | W      |
                                         654|       | T        |   I                | |        | |W V     | | W      |
 IS  X     V0  ,A0,A0           130B     655|X      | T        |               V    | |        | |W V     | | W      |         X
                                         656|X      |          |               V    | |        | |W V     | | W      |         X
                                         657|X      |          |               V    | |       |W|W V     | | W      |         X
                                         658|X      |          |               V    | |        | |  V     | | W      |         X
                                         659|X      |          |                    | |        | |  V     | | W      |         X
                                         660|X      |          |                    | |        | |  V     | |        |         X
                                         661|X      |          |                    | |       |V|  V     | |        |         X
                                         662|X      |          |                    | |        | |        | |        |         X
```

Figure 2.5.5 - 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 pipelined. 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.

2.5.2 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.

2.5.2.1 Scalar Example

The CPACT report shown in figure 2.5.5 (wide and narrow versions) was produced 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 is first started up, no instruction parcels are in the parcel buffers so a fetch is 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.

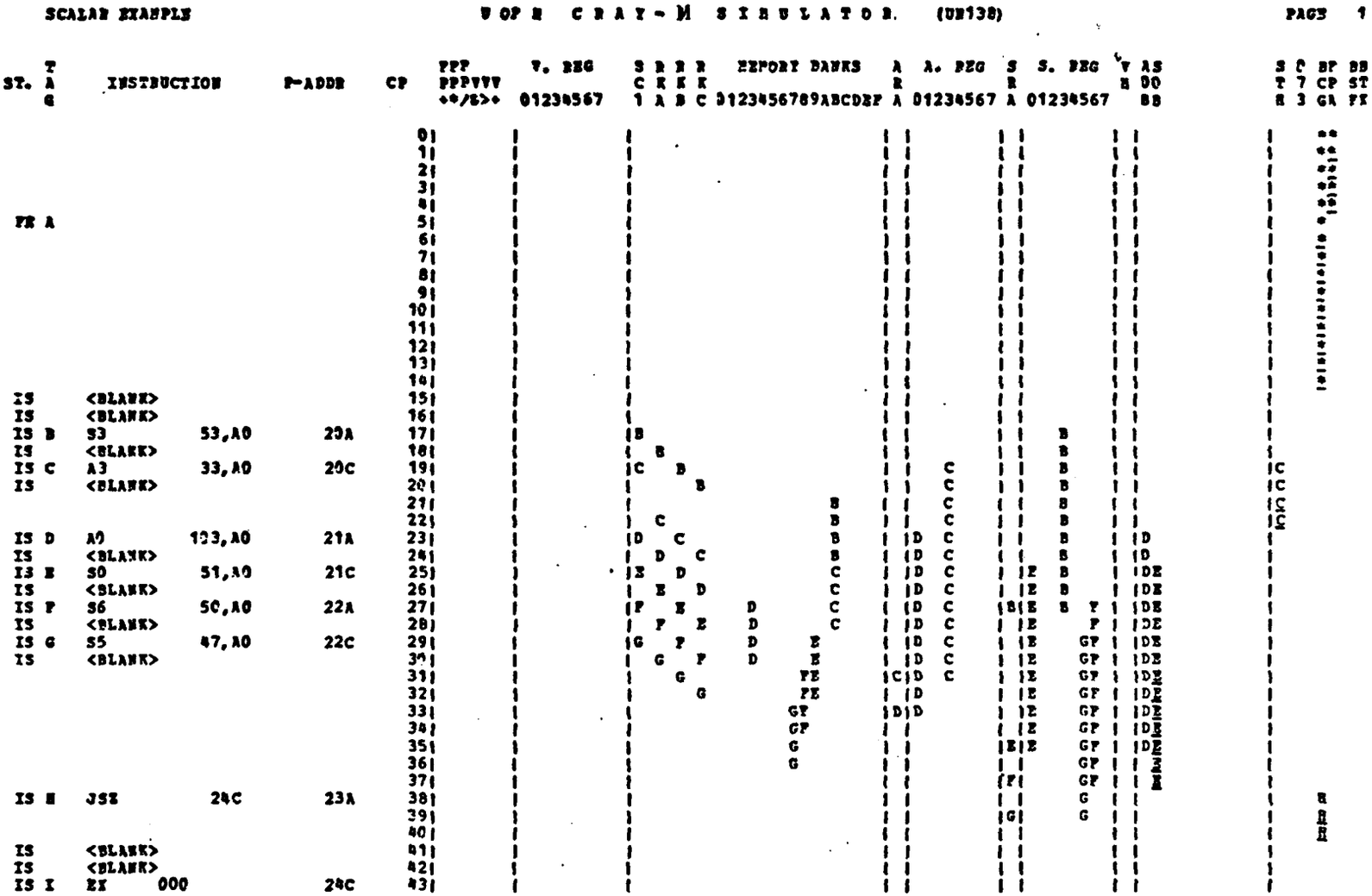


Figure 2.5.6 - CPACT Scalar Example, Wide version

```
S T                                    FFP      V. REG   BSRRR S S. REG   A A. REG   SOBF
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 |       |CC
                                   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 |       |
1   <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  |          |       |HH
                                   40|       |         |     |       |  |          |       |
I   <BLANK>                        41|       |         |     |       |  |          |       |
I   <BLANK>                        42|       |         |     |       |  |          |       |
EXIT    0 AT      24C
.
```

Figure 2.5.6 - 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.

2.5.2.2 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.

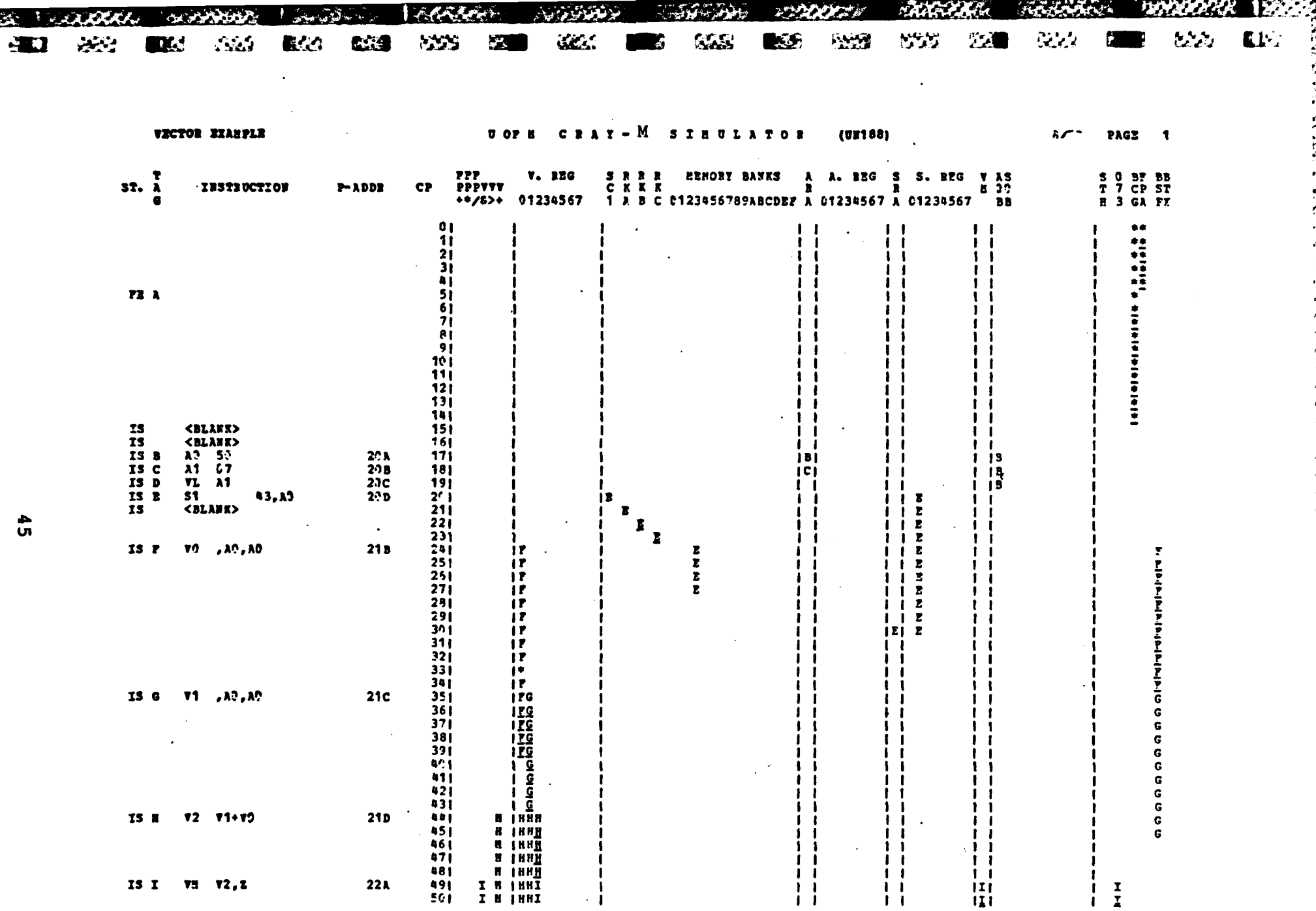


Figure 2.5.7 - CPACT Vector Example

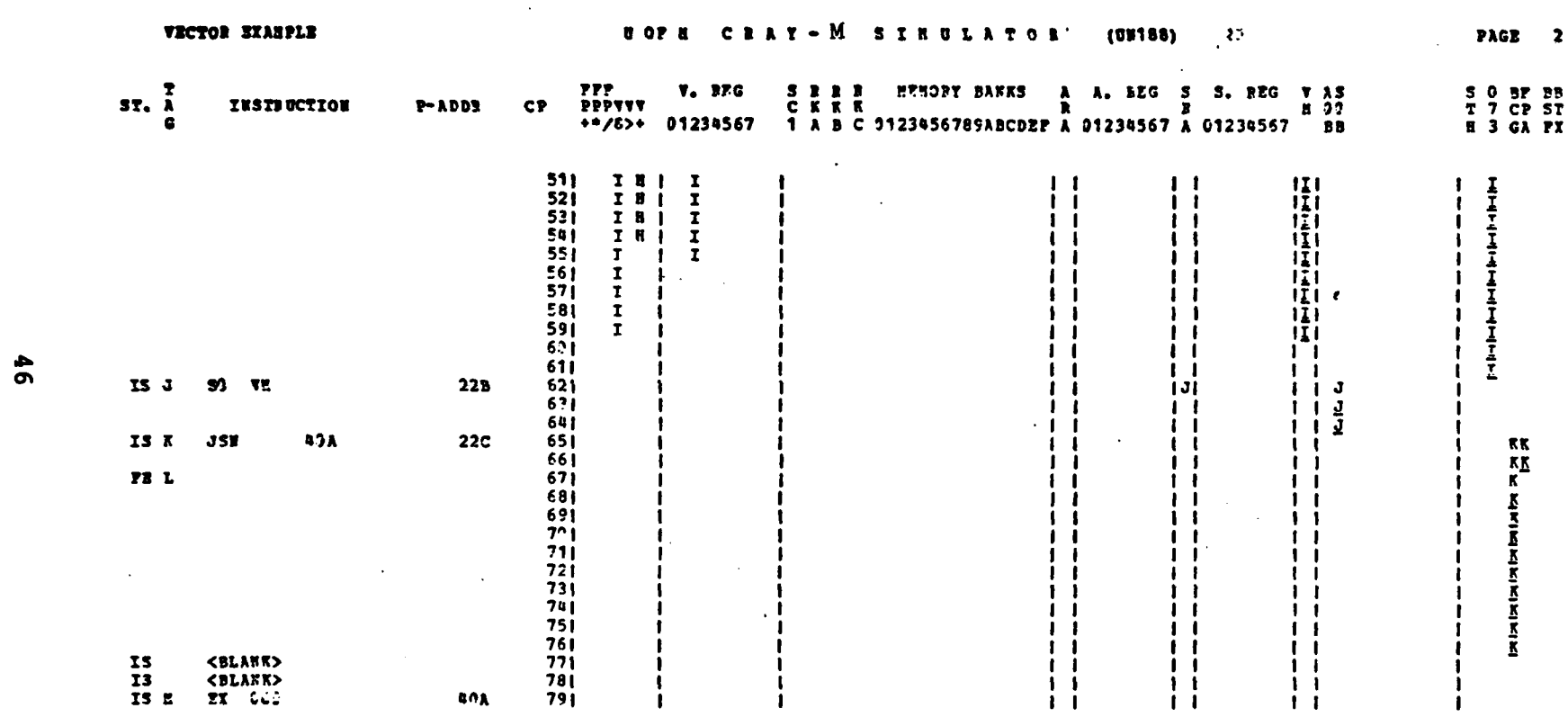


Figure 2.5.7 - CPACT Vector Example "continued"

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.

2.5.3 TRACE Report

The TRACE report is an instruction-by-instruction flowtrace of the program being executed. As each instruction issues, its instruction affects are displayed. The TRACE report can be enabled or disabled for any or all of the CPU's. The TRACE report, unlike the STAT report, is independent of the TIMING switch. The format of the TRACE command, which is somewhat complicated, will not be discussed here -- see instead Chapter 3 of this report. Here will be presented some examples of the use of the TRACE command, the sample code used is the same as was used in the STAT example (section 2.5.1.1).

The TRACE report produces, for each instruction, one or more printed lines. The output for each instruction is divided into three fields: address, mnemonic and display; the fields are separated by colons. The address field contains the parcel address of instruction, and the mnemonic field contains the CAL (Cray-1 Assembly Language) instruction mnemonic of the instruction. Thus, looking at (1) in the figure, we see that at parcel address 20B is the CAL instruction A2 7750, A0 (opcode 10h). Note that all constants displayed in both address and mnemonic fields are octal constants, regardless of what BASE pseudo-op was used for assembly of the source code. Therefore, since $7750_8 = 4072_{10}$, the memory reference will not exceed the address space (4096_{10} words) provided that $-4072_{10} \leq A0 \leq 23_{10}$.

The third field in the TRACE output line (or lines) is the display field. In the display field is printed the contents of storage locations affected by the instruction. The results displayed are results <u>after</u> the instruction has exectued; that is, the results which the instruction produced. In (1), this means that A0=0 and that the word stored in A2 after being fetched from memory is also zero. For every instruction, the result register is displayed unless the instruction is a cache transfer instruction (opcodes 025 and 075), in which case the contents of the A or S register transferred into the B or T cache is displayed rather than the actual S or T result register. As an example, see (2); here the contents of A4 rather than the contents of B05 are displayed.

All displayed results are decimal integers or decimal floating point numbers. In (3), the result is shown to be 2023_{10}. Exceptions to the decimal-display rule are the following:

1) S registers are displayed as both decimal integers and decimal floating-point numbers unless the instruction is a logical instruction (shift, mask, AND, etc.) in which case the result is displayed as a 64-bit octal constant.

2) V registers are displayed as octal constants for logical instructions, and decimal floating-point numbers otherwise (contrast (4) and (5)).

Vector instructions display a number of elements which is equal to the minimum of VL and the number specified by the optional LEN parameter on the TRACE command. Note that in (4) and (5) all 64 elements are printed since VL=64 and no LEN parameter was supplied. (6) illustrates a TRACE command with a LEN parameter given; since a LEN is given, (7) (which is the same instruction (4)) displays only four elements of the vector result register.

```
. MAP
 MODULE          LOCATION      LENGTH
MMUL                  20A         202

. TRACE ON
. RU      #25
     20A :  A0  00            :   A0= 0
 →   20B :  A2      7750,A0   :   A0= 0  A2= 0 ←— (1)
     20D :  A4      7751,A0   :   A0= 0  A4= 0
 →   21B :  B05 A4            :   A4= 0 ←— (2)
     21C :  A4  A4*A2          :   A4= 0
 →   21D :  A7       3747     :   A7= 2023 ←— (3)
     22B :  B02 A7             :   A7= 2023
     22C :  A7       5747     :   A7= 3047
     23A :  B04 A7             :   A7= 3047
     23B :  A5  01             :   A5= 1
     23C :  B20 A5             :   A5= 1
     23D :  VL  A4             :   VL=64
     24A :  A1  00             :   A1= 0
     24B :  B07 A1             :   A1= 0
 →   24C :  V3  S0&V4          :   VL=64 ←— (4)
 V3( 0)= 0'0'  V3( 1)= 0'0'  V3( 2)= 0'0'
 V3( 3)= 0'0'  V3( 4)= 0'0'  V3( 5)= 0'0'
 V3( 6)= 0'0'  V3( 7)= 0'0'  V3( 8)= 0'0'
 V3( 9)= 0'0'  V3(10)= 0'0'  V3(11)= 0'0'
 V3(12)= 0'0'  V3(13)= 0'0'  V3(14)= 0'0'
 V3(15)= 0'0'  V3(16)= 0'0'  V3(17)= 0'0'
 V3(18)= 0'0'  V3(19)= 0'0'  V3(20)= 0'0'
 V3(21)= 0'0'  V3(22)= 0'0'  V3(23)= 0'0'
 V3(24)= 0'0'  V3(25)= 0'0'  V3(26)= 0'0'
 V3(27)= 0'0'  V3(28)= 0'0'  V3(29)= 0'0'
 V3(30)= 0'0'  V3(31)= 0'0'  V3(32)= 0'0'
 V3(33)= 0'0'  V3(34)= 0'0'  V3(35)= 0'0'
 V3(36)= 0'0'  V3(37)= 0'0'  V3(38)= 0'0'
 V3(39)= 0'0'  V3(40)= 0'0'  V3(41)= 0'0'
 V3(42)= 0'0'  V3(43)= 0'0'  V3(44)= 0'0'
 V3(45)= 0'0'  V3(46)= 0'0'  V3(47)= 0'0'
 V3(48)= 0'0'  V3(49)= 0'0'  V3(50)= 0'0'
 V3(51)= 0'0'  V3(52)= 0'0'  V3(53)= 0'0'
 V3(54)= 0'0'  V3(55)= 0'0'  V3(56)= 0'0'
 V3(57)= 0'0'  V3(58)= 0'0'  V3(59)= 0'0'
 V3(60)= 0'0'  V3(61)= 0'0'  V3(62)= 0'0'
 V3(63)= 0'0'
 →  24D :  V0  S0+FV3       :   VL=64 ←— (5)
 V0( 0)= 0.0          V0( 1)= 0.0          V0( 2)= 0.0
 V0( 3)= 0.0          V0( 4)= 0.0          V0( 5)= 0.0
 V0( 6)= 0.0          V0( 7)= 0.0          V0( 8)= 0.0
 V0( 9)= 0.0          V0(10)= 0.0          V0(11)= 0.0
 V0(12)= 0.0          V0(13)= 0.0          V0(14)= 0.0
 V0(15)= 0.0          V0(16)= 0.0          V0(17)= 0.0
 V0(18)= 0.0          V0(19)= 0.0          V0(20)= 0.0
 V0(21)= 0.0          V0(22)= 0.0          V0(23)= 0.0
 V0(24)= 0.0          V0(25)= 0.0          V0(26)= 0.0
 V0(27)= 0.0          V0(28)= 0.0          V0(29)= 0.0
 V0(30)= 0.0          V0(31)= 0.0          V0(32)= 0.0
 V0(33)= 0.0          V0(34)= 0.0          V0(35)= 0.0
 V0(36)= 0.0          V0(37)= 0.0          V0(38)= 0.0
 V0(39)= 0.0          V0(40)= 0.0          V0(41)= 0.0
```

Figure 2.5.8. TRACE Example

```
V0(42)= 0.0          V0(43)= 0.0          V0(44)= 0.0
V0(45)= 0.0          V0(46)= 0.0          V0(47)= 0.0
V0(48)= 0.0          V0(49)= 0.0          V0(50)= 0.0
V0(51)= 0.0          V0(52)= 0.0          V0(53)= 0.0
V0(54)= 0.0          V0(55)= 0.0          V0(56)= 0.0
V0(57)= 0.0          V0(58)= 0.0          V0(59)= 0.0
V0(60)= 0.0          V0(61)= 0.0          V0(62)= 0.0
V0(63)= 0.0
   25A :  A6  B02          :   A6= 2023
   25B :  A6  A6+A4        :   A6= 2023
   25C :  A7  A0-A0        :   A7=-1
   25D :  A0  A0+A6        :   A0= 2023
   26A :  V1  ,A0,A7       :   A0= 2023  A7=-1  VL=64
V1( 0)= 0.0          V1( 1)= 0.0          V1( 2)= 0.0
V1( 3)= 0.0          V1( 4)= 0.0          V1( 5)= 0.0
V1( 6)= 0.0          V1( 7)= 0.0          V1( 8)= 0.0
V1( 9)= 0.0          V1(10)= 0.0          V1(11)= 0.0
V1(12)= 0.0          V1(13)= 0.0          V1(14)= 0.0
V1(15)= 0.0          V1(16)= 0.0          V1(17)= 0.0
V1(18)= 0.0          V1(19)= 0.0          V1(20)= 0.0
V1(21)= 0.0          V1(22)= 0.0          V1(23)= 0.0
V1(24)= 0.0          V1(25)= 0.0          V1(26)= 0.0
V1(27)= 0.0          V1(28)= 0.0          V1(29)= 0.0
V1(30)= 0.0          V1(31)= 0.0          V1(32)= 0.0
V1(33)= 0.0          V1(34)= 0.0          V1(35)= 0.0
V1(36)= 0.0          V1(37)= 0.0          V1(38)= 0.0
V1(39)= 0.0          V1(40)= 0.0          V1(41)= 0.0
V1(42)= 0.0          V1(43)= 0.0          V1(44)= 0.0
V1(45)= 0.0          V1(46)= 0.0          V1(47)= 0.0
V1(48)= 0.0          V1(49)= 0.0          V1(50)= 0.0
V1(51)= 0.0          V1(52)= 0.0          V1(53)= 0.0
V1(54)= 0.0          V1(55)= 0.0          V1(56)= 0.0
V1(57)= 0.0          V1(58)= 0.0          V1(59)= 0.0
V1(60)= 0.0          V1(61)= 0.0          V1(62)= 0.0
V1(63)= 0.0
   26B :  B02 A6           :   A6= 2023
   26C :  A3  01           :   A3= 1
   26D :  A7      1747     :   A7= 999
   27B :  B03 A7           :   A7= 999
** INSTRUCTION ISSUE LIMIT EXCEEDED AT      27C
. TRACE OFF
. TRACE ON    LEN=4 <-- (6)
. RU 20A #25
   20A :  A0  00           :   A0= 0
   20B :  A2      7750,A0  :   A0= 0  A2= 0
   20D :  A4      7751,A0  :   A0= 0  A4= 0
   21B :  B05 A4           :   A4= 0
   21C :  A4  A4*A2        :   A4= 0
   21D :  A7      3747     :   A7= 2023
   22B :  B02 A7           :   A7= 2023
   22C :  A7      5747     :   A7= 3047
   23A :  B04 A7           :   A7= 3047
   23B :  A5  01           :   A5= 1
   23C :  B20 A5           :   A5= 1
   23D :  VL  A4           :   VL=64
   24A :  A1  00           :   A1= 0
   24B :  B07 A1           :   A1= 0
```

Figure 2.5.8. TRACE Example (cont'd)

```
--> 24C :  V3   S0↑V4       :   VL=64 <-- (7)
  V3( 0)= 0'0'  V3( 1)= 0'0'  V3( 2)= 0'0'
  V3( 3)= 0'0'
    24D :  V0   S0+FV3      :   VL=64
  V0( 0)= 0.0           V0( 1)= 0.0           V0( 2)= 0.0
  V0( 3)= 0.0
    25A :  A6   B02         :   A6= 2023
    25B :  A6   A6+A4       :   A6= 2023
    25C :  A7   A0-A0       :   A7=-1
    25D :  A0   A0+A6       :   A0= 2023
    26A :  V1   ,A0,A7      :   A0= 2023  A7=-1  VL=64
  V1( 0)= 0.0           V1( 1)= 0.0           V1( 2)= 0.0
  V1( 3)= 0.0
    26B :  B02  A6          :   A6= 2023
    26C :  A3   01          :   A3= 1
    26D :  A7       1747    :   A7= 999
    27B :  B03  A7          :   A7= 999
** INSTRUCTION ISSUE LIMIT EXCEEDED AT      27C
. TRACE OFF
```

Figure 2.5.8. TRACE Example (cont'd)

2.5.4 TACT STAT Report

The Task ACTivity STATistics report is a condensed table of all tasking activity since tasking was turned on (SET TASK=ON). Each column of the report shows how much time (in clock periods) each CPU spent executing each task. The percentage of the total time since timing was turned on is shown beneath the number of clock periods. The last column, labelled 'TOT', is the total amount of processor time spent in each task.

Each row of the report shows the time each processor spent executing a particular task. The last row depicts the time each CPU spent executing the tasks. The last entry in the last row represents the overall task concurrency. If all the CPU's spent all of their time executing defined tasks, this figure would be the number of clock periods multiplied by the number of CPU's. For example, if three CPU's spent forty percent of their time executing defined tasks, the total concurrency percentage would be %120 out of a total possible of %300.

Figure 2.5.7.1 is a TACT STAT report from simulation of a four-processor sparse matrix triangular factorization.

TASK STATISTICS

TASK	CPU 1	CPU 2	CPU 3	CPU 4	TOT
FAC1	86	0	0	0	86
	1.28%	0.0 %	0.0 %	0.0 %	1.28%
JOIN1	369	0	0	0	369
	5.50%	0.0 %	0.0 %	0.0 %	5.50%
MUL2	0	0	0	0	0
	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %
JOIN4	18	0	0	0	18
	0.27%	0.0 %	0.0 %	0.0 %	0.27%
JOIN5	0	0	0	0	0
	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %
SOL1	325	0	0	0	325
	4.84%	0.0 %	0.0 %	0.0 %	4.84%
MUL1	0	0	0	0	0
	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %
MUL	0	0	0	0	0
	0.0 %	0.0 %	0.0 %	0.0 %	0.0 %
SOL	1310	1310	1310	1310	5240
	19.52%	19.52%	19.52%	19.52%	78.08%
FAC	0	380	0	0	380
	0.0 %	5.66%	0.0 %	0.0 %	5.66%
TOTAL	2108	1690	1310	1310	6418
	31.41%	25.18%	19.52%	19.52%	95.63%

TOTAL CLOCKS: 6711

Figure 2.5.7.1: Sample TACT STAT Report

2.5.5 The TACT Report

A task is an identified group of instructions. It must have unique entry and exit point. A name is associated with each task that is defined. When a processor passes through a task entry point, the name of the task is placed in that processor's column on the TACT report. The name of the task remains in the processor's column until the processor passes through the exit point of that task, at which point the column entry is cleared.

The Task Activity Report is a detailed listing of all tasking activity, similar to the CPACT Report. The TACT Report can be thoughtof as a macro CPACT Report, the CPU's representing functional units and tasks representing instructions.

To enable task information collection, the command "SET TASK=ON" is issued from the command language. The simulator then prompts for a file containing the task definitions for the programs being simulated (for a sample task definition file, see Appendix K). To enable TACT Reporting, the command "TACT filename" is then issued where "filename" is the name of the file to receive the TACT report output. The program is then run in the usual fashion.

The sample report shown in Figure 2.5.7.2 is for a blocked randomly sparse symmetric matrix factorization.

```
                                TASK ACTIVITY REPORT

                                    8X8 SPARSE


 CP     CPU 1   CPU 2   CPU 3   CPU 4
-------------------------------------
   .                       .
   .                       .
   .                       .
 2150|       |       |       |       |
 2200|JOIN4  |       |       |       |
 2218|       |       |       |       |
 2250|       |       |       |       |
 2350|       |       |       |       |
 2389|FAC1   |       |       |       |
 2450|FAC1   |       |       |       |
 2475|       |       |       |       |
 2515|       |FAC    |       |       |
 2550|       |FAC    |       |       |
 2573|JOIN1  |FAC    |       |       |
 2650|JOIN1  |FAC    |       |       |
 2750|JOIN1  |FAC    |       |       |
 2850|JOIN1  |FAC    |       |       |
 2895|JOIN1  |       |       |       |
 2942|       |       |       |       |
 2943|SOL1   |       |       |       |
 2950|SOL1   |       |       |       |
 3050|SOL1   |       |       |       |
 3060|       |       |       |       |
 3090|       |SOL    |       |       |
 3150|       |SOL    |       |       |
 3175|SOL1   |SOL    |       |       |
 3250|SOL1   |SOL    |       |       |
 3255|       |SOL    |       |       |
 3295|       |SOL    |SOL    |       |
 3350|       |SOL    |SOL    |       |
 3361|SOL1   |SOL    |SOL    |       |
 3427|       |SOL    |SOL    |       |
 3450|       |SOL    |SOL    |       |
 3468|       |SOL    |SOL    |SOL    |
 3533|SOL1   |SOL    |SOL    |SOL    |
 3550|SOL1   |SOL    |SOL    |SOL    |
 3595|       |SOL    |SOL    |SOL    |
 3650|       |SOL    |SOL    |SOL    |
 3750|       |SOL    |SOL    |SOL    |
 3850|       |SOL    |SOL    |SOL    |
 3950|       |SOL    |SOL    |SOL    |
 3990|SOL    |SOL    |SOL    |SOL    |
 4050|SOL    |SOL    |SOL    |SOL    |
 4150|SOL    |SOL    |SOL    |SOL    |
 4250|SOL    |SOL    |SOL    |SOL    |
 4350|SOL    |SOL    |SOL    |SOL    |
 4400|SOL    |       |SOL    |SOL    |
 4450|SOL    |       |SOL    |SOL    |
 4550|SOL    |       |SOL    |SOL    |
 4605|SOL    |       |       |SOL    |
 4650|SOL    |       |       |SOL    |
 4750|SOL    |       |       |SOL    |
 4778|SOL    |       |       |       |
 4850|SOL    |       |       |       |
 4950|SOL    |       |       |       |
 5050|SOL    |       |       |       |
 5150|SOL    |       |       |       |
 5250|SOL    |       |       |       |
 5300|       |       |       |       |
```

Figure 2.5.7.2 Sample TACT Report

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
5) The simulator floating point format (IBM 360/370) differs from the Cray-1 format. (See section 2.1.3)
6) The 071X2X instruction (Si +AK) produces a normalized result in the simulator. Not so on the Cray-1.
7) 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. /

3. 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.

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.

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 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	
TRACE	ON|OFF [fdname] [LEN = VL|trace length]
USE	fdname [NOECHO]
$MTS-command	
CPU	CPU number
ENABLE	CPU list
DISABLE	CPU list
TACT	STAT|fdname

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. A maximum of 9 commands can be accepted. 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 S0 A0 M(33)
END
```

When the AT point is hit, registers S0 A0 and memory location 33 octal are displayed.

```
AT 245B 31
D M(0),10
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. Befor the instruction is executed again the AT point takes control and displays memory locations throcugh 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 th 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.

CPU
Command Description

Purpose: To specify the indicated CPU as the current CPU

Prototype: CPU cpu number

Description:

The CPU command is used to change the current cpu to the specified cpu. The current cpu is the cpu to which all commands issued apply (specifically TRACE, CPACT, DISPLAY and CHANGE). The CPU command can be thought of as a global scope specifying command. Instead of specifying to which CPU each command pertains, a global CPU number is specified, and all subsequent applicable commands pertain to the specified CPU.

To enable instruction tracing on CPU 3, the commands "CPU 3" and "TRACE ON" are given. To change the program counter of CPU 1 (i.e., prior to a RUN command) the commands "CPU 1" and "CH P MAIN" are given.

Examples:

```
CPU 2
CPU 4
```

Error responses:

Invalid CPU number -

Issued when an incorrect value is specified for the CPU number.

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 specifying W, P, or V. If this is done, the symbol will be defined as type word address, parcel address, or value, respectively.

The symbol is added to the symbol table of the current IDENT; if there is no governing IDENT, then an error will result. The user must issue at least one IDENT command before using the DEFINE command.

Examples:

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

DISABLE
Command description

Purpose: To deactivate a cpu.

Prototype: DISABLE cpu list

Description:

The DISABLE command is used to "turn off" a cpu. Any cpu except the current cpu can be specified by the command.

Examples:

```
DISABLE 2 3 4
DISA 3
```

Error responses:

Cannot disable current cpu -
: issued when trying to disable the cpu last specified by the CPU command

Invalid cpu number -
: Issued when an incorrect value is specified in the CPU list.

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 facility 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.

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
RF	Relocation factor	No	Yes
FLAGS	FLAGS register	No	No
LA	Limit address register	No	No
BA	Base address register	No	No
MSIZ	Value of memory size	No	No
MODE	Mode register	No	Yes

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  A0,8
D@PI  IM(p),10 MAIN, LEN$
D@EFO  V0(0),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.

ENABLE
Command description

Purpose: To activate other cpu's for multi-tasking

Prototype: ENABLE cpu list

Description:

The ENABLE command is used to "turn on" a cpu. The RUN command applies to all cpu's activated by the ENABLE command. Any or all of the cpu's can be specified by the activate command. Up to four cpu's (1, 2, 3, 4) can be enabled in the current version of the simulator; this can readily be changed in the source code.

Examples:

```
ENABLE 2 3 4
ENA 3 4
```

Error response:

cpu already enabled -
Issued when a cpu specified in the list is already activated

Invalid cpu number -
Issued when an invalid number is specified in the cpu list

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
```

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 [STAT]

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 is initialized.
3. The simulator state is cleared.
4. The CPU clock is cleared.
5. The CIP, NIP, LIP, and instruction buffers are invalidated.

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

If the STAT parameter is specified, then only the timing information and the CPU clock are initialized.

Examples:

```
INIT
I STAT
```

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

Description :

The MAP command will display the names, starting locations, and lengths of all loaded modules.

Examples:

MAP

MA

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 tables.

Prototype: REMOVE symbol

Description:

The specified symbol is removed from the symbol table of the current IDENT (set via the IDENT command). If an IDENT command has not been previously given, the issue of a REMOVE command will cause an error.

Examples:

```
REMOVE  START
REM  ARRAY1
```

Error Response:

symbol is not defined.

RETURN

Command description

Purpose : To allow the simulator to return to its caller.

Prototype : RETURN

Description:

The RETURN command is used to force the simulator to return to its caller. Normally, when the simulator is called as a subroutine, the CRAY-1 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
```

RUN
Command Description

Purpose: To begin simulation of a Cray-M assembly program

Prototype: RUN [#issue limit]

Description:

The RUN command is the only simulator command that actually begins the simulation. All active CPU's begin execution at their current program counter locations.

Before the initial RUN command is given, the program counters of all the active cpu's must be given initial values. The starting location must be specified by using the CHANGE command:

```
CPU cpu number
CHANGE P <Start address>
```

This alters the program counter for the specified cpu. An error message is issued if an active CPU has an uninitialized program counter when a RUN command is given.

The optional issue limit parameter can be used to control the simulation. 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.

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 procedure to stop a program. Other common conditions that stop simulation are breakpoints, at-points, or issue limit expired. The exceptional conditions that halt simulation are discussed in section 2.2.

Examples:

```
Run #5000
RUN
R
R #1
```

The last example illustrates how the user would single step through the program, executing one instruction at a time.

Error response:

CPU program counter not set -

Issued when a CPU's programcounter has not been initialized.

Invalid issue limit -

Issued when an incorrect issue limit is specified.

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
TASK	ON, OFF	OFF

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.

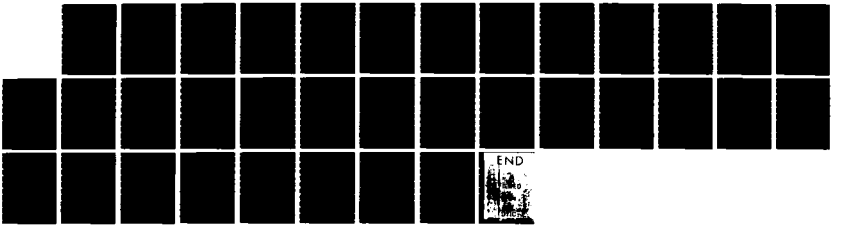
OUTPUT Default: *MSINK*

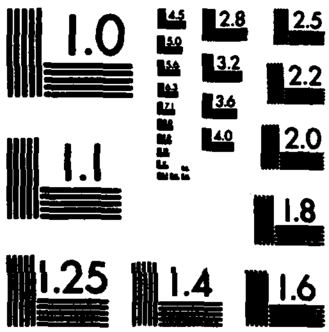
The OUTPUT keyword controls the file or device to which the simulator sends all normal output (i.e., not prompts or error messages, which always go to the terminal). Normal output includes informational messages, DISPLAY, HELP, and STAT output. When the simulator starts up OUTPUT is set to *MSINK* (the 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.

AD-A136 555 A CRAY-CLASS MULTIPROCESSORSSIMULATOR(U) MICHIGAN UNIV 2/2
ANN ARBOR SUPECCOMPUTERAALGORITHM RESEARCH LAB
P M SUMMERS ET AL. 01 SEP 83 SARL-1 AFOSR-RR-83-1246
UNCLASSIFIED AFOSR-80-0158 F/G 9/2 NL





TASK

The TASK keyword controls gathering of task statistics. Setting TASK equal to ON also enables resource timing (TIM=ON), which increases simulation cost by a factor of 8 to 10. The default state of TASK is OFF. When TASK is turned on, the simulator prompts for the name of a file containing task definitions. The contents of the file control the format of the TACT Report as well as the definition of tasks is simulator memory.

A task is a section of code that has a unique entry and exit point. When a cpu enters the task, the task's name is entered that cpu's column in the TACT Report. When the cpu passes through the exit point, the task name is removed from the cpu's column. Upon entry and exit from a task, timing information is recorded for later use in the TACT STAT report. For a detailed description of the Task Definition file, see Appendix K.

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.

TRACE

Command Description

Purpose : To control the generation of the trace output report, a report of data transfers for each instruction.

Prototype : TRACE ON|OFF [fdname] [LEN = VL|trace length]

Description:

The TRACE command enables and disables the trace output report. The trace output report consists of the instruction parcel address, instruction mneumonic, and the contents of relevent registers. If fdname is not specified the output is sent to the fd specified by the SET OUTPUT = fd command (the default is *MSINK*).

When "LEN = VL" is specified all results produced by vector operations will be displayed. In the case of B and T block transfers all elements transferred will be displayed. If "LEN = n" ($0 \leq n \leq 64_{10}$) is specified n elements are displayed on vector operations. The minimum of n and the block transfer length will be displayed for B and T block transfers. (The default value is LEN = VL).

The trace length may also be set using the SET command: "SET LEN = trace length". The following page shows a trace output example.

EXAMPLES:

```
TRACE ON
T ON      -A    L=1Ø
T ON      L=VL
T ON      LEN=2Ø
T OFF
```

ERROR MESSAGES:

```
ERROR - INVALID RIGHT HAND SIDE:  rhs (e.g. LEN = - 1)
ERROR - INVALID LEFT HAND SIDE:   lhs (e.g. LENT = 10)
*** INVALID   TRACE COMMAND   PARAMETERS
*** INVALID   TRACE COMMAND   FDNAME
```

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 c mands. 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.

4. Cray-M Simulation Costs

Because instruction-level simulation is admittedly costly, it is important to utilize only the level of simulation (numerical versus timing) and the reporting level appropriate to the need. Fortunately, the interactive nature of the simulator makes it possible to switch these levels on and off during a run; the most costly levels are rarely required for an entire simulation.

Table 1 gives the costs of running 1000-5000 instructions with a variety of simulation and reporting levels. (Note that semaphores and shared registers rotate without timing on (see Appendix E)). Among the figures given, the most significant appears to be

(a) a 8970:1 speedown of uniprocessor CRAY-1 time to Amdahl time; the simulation costs increase approximately linearly with the number of processors.

(b) a 3.3:1 ratio of costs between simulation with timing on and timing off, per processor; this ratio has been found to be as high as 5:1 for highly vectorized code.

As a benchmark case, a million clock, 4-processor run costs approximately $100 at minimum rates (4-7 am) and $500 at regular rates.

	Number of Processors			Units
	1	2	4	
TIM=OFF	1.52	1.32	1.30	¢/kiloinstruction*
TIM=ON	2.07	3.88	9.23	¢/kiloclock
	8970	17200	37500	Amdahl time/CRAY-1 time
TRACE ON$^{+\Delta}$	88	63	46	¢/kiloinstruction*
CPACT$^{\Delta}$	72	130	250	¢/kiloclock

* Instructions summed across all processors

\+ TIM=OFF

Δ Printing costs not included

Table 1. Simulation costs; minimum rates used (20% normal rates), approximately 19¢/sec; approximately 2.4 clocks/(instruction issue) (33 MIPS) per processor in code used.

5. Bibliography

1) A CRAY-1 Simulator, publication No. 118, Systems Engineering Laboratory, University of Michigan by D. A. Orbits, 1978.
2) Cray-1 Hardware Reference, publication No. 2240004, by Cray Research Inc., 1980.
3) Cray-1 Fortran Reference Manual, publication no. 2240009, by Cray Research Inc., 1980.
4) Cray-1 External Reference Specification, publication no. 2240011, by Cray Research Inc., 1976.
5) Cray-1 Fortran Mathematical Subprogram Library Reference Manual, publication no. 2240014, by Cray Research Inc., 1977.
6) Cray-1 Reference Card, publication no. SQ-0003, by Cray Research Inc., 1981.
7) Cray-1 CAL Assembler Reference Manual, publication no. SR-0000, 1980.
8) Introduction to Vector Processing on the Cray-1 Computer, publication no. 2240002, 1975.
9) The Cray-1 Computer System, CACM, by Richard M. Russell, 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 placé 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 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.

Functional unit	Time (c.p.)
Logical	2
Shift	4
Integer add	3
Floating add	6
Floating multiply	7
Reciprocal approximation	14
Memory	7

† Multiple of 4 for 8-bank phasing; refer to section 5.

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.

† 18 clock periods for 8-bank phasing option; refer to section 5.

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 NIP 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 NIP 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 destination register 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 and TACT output uses I/O units 30 through 64.
- 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, 2Ø and 30 through 64.

Appendix C.
Cray-1 Simulator Common Block Usage

The Cray-1 simulator currently uses 30 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
BRKCOM
COM$F
CONTRL
CTABLE
DECTBL
DEV
DRVTBL
INSTRX
LIP
LOAD
MEMORY
MSFLAG
MSIZE
QCODES
QCOM
REG
REPORT
SETABL
STATE
SYMTB1
SYMTB2
TRAPAR
TRKBLK
UNITS
USAGE
XCHANG
NEW
TASKS
$

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-M instructions to simulate shared registers and semaphores *

In developing the CRAY-M simulator, we decided that some means of close communication between the processors should be provided. We therefore added eight shared T registers, eight shared B registers and sixty-four semaphore registers. There are three instructions for manipulating 64 semaphore registers, two instructions for the shared T registers and two instructions for the shared T registers and two instructions for the shared B registers.

To avoid conflict, access to the semaphores and shared registers "rotates" between the active CPU's. This rotation is based on the Real Time clock register when timing is enabled, and on the number of instructions issued when timing is disabled. It should be noted that this difference in rotation methods may cause different results in tightly coupled algorithms.

All timings and protocol (such as rotation and the phasing of shared registers) are the author's choices and do not necessarily reflect behavior of a product of Cray Research, Inc.

SMjk	0	Clear semaphore jk. Semaphore register jk is set to 0. Instruction will take from 1 to 4 clocks to complete.
SMjk	1	Set semaphore jk. Semaphore register jk is set to 1. Instruction will take from 1 to 4 clocks to complete.
SMjk	1,TS	Test and set semaphore jk. If semaphore register jk is 0, set it to 1 and continue. If semaphore register jk is 1, hold issue on this instruction (i.e., until a different cpu sets the semaphore to 0).
SJ	STk	Enter Sj with STk (shared T register k). This instruction will take from 1 to 4 clocks to complete, but is phased to execute immediately following a semaphore instruction.

STj	Sk	Enter STj (shared T register j) with Sk. This instruction will take from 1 to 4 clocks to complete, but is phased to execute immediately following a semaphore instruction.
Aj	SBk	Enter Aj with SBk (shared B register k). This instruction will take from 1 to 4 clocks to complete, but is phased to execute immediately following a semaphore instruction

Appendix F

Cray-M Simulator Error Stops

This appendix discusses possible simulator error stops. These error stops are caused by internal simulator errors that could adversely affect simulation results if simulation were allowed to proceed.

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

Error Stops with Messages

Subroutine	Stop Message
GETCMD	END OF FILE ON BATCH INPUT STREAM.
DECODE	INTERNAL ERROR. DECODE TABLES CLOBBERED.
SIMBRK	SIMBRK CALLED BUT BRKSET .LE. ZERO.
SIMBRK	SIMBRK CALLED BUT BREAKPOINT NOT IN TABLE.
WINST	INTERNAL ERROR. DECODE TABLES CLOBBERED.

Error Stops with Stop Codes

Subroutine	Stop Code	Comments
MSW	101	Invalid bit code in MSW.
MSW	102	Invalid argument to MSW.
SMCTRL	103	Unimplemented action code used.
SMCTRL	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.
WINST	113	Bad instruction format code.
SMCTRL	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-M Simulator

Language: IBM Fortran-IV
IBM Assembly Language

Operating System Requirements: The only system subroutines needed are those provided by the standard IBM FORTRAN IV Subroutine Library (e.g., MAX0, MIN0, etc.). All I/O is done via FORTRAN READ and WRITE statements with record lengths of 80 bytes or less. Hence, the simulator should run on any IBM-based operating system.

Availability: Source code for the simulator is available on 9-track EBCDIC tapes. Tapes can be made according to any blocking format, can be labelled or unlabelled, and can be made at 800, 1600, or 6250 BPI. The entire Simulator/Cross-Assembler package is approximately 300,000 bytes long.

Contact:

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

Appendix H.

Sample Simulator Exit Dispatcher

```
      SUBROUTINE CRAYEX(IJK, ASP, SSR, VSR, VL, EXSW)
      IMPLICIT INTEGER(A-Z)
      LOGICAL EXSW, NOANS/.TRUE./
      INTEGER ASP(8), RTC(2)
      REAL*8 SSR(8), VSR(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=ASP(2+1)
      MADDR=ASP(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
```

```
C
C
C ... CODE=2 PRINT THE TIME.  (A7 = MATRIX SIZE, S7 = RUN TIME)
200   DS = SYR(7+1)
      WRITE(6,1000) ASR(7+1), RTC(2)
1000  FORMAT(' SIZE=',I5,'  RTC=',I7)
C     SXSW = .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 I.

Sample Simulator Calling Program

```
      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.)
```

```
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.)
```

```
C
C ... SET UP THE ARGUMENT CALL BLOCK WITH POINTERS TO
C     THE ARGUMENTS.
C
C    LOC 100 = NSYS,  LOC 101 = NEQS,  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) = NEQS
      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 + (NEQS-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 * ((NEQS-1)*2 + NEQS*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



      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
Load Module Formats

1. 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 five spaces, 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 five spaces 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, followed 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.

APPENDIX K

Task Definition File Description

By issuing the command "SET TASK=ON" from the simulator command language, task timing is enabled. The simulator immediately prompts for an input file defining the task locations in simulator memory.

The structure of the input file is as follows:

<TACT Report Header, 1 to 40 characters>
<Clock skip> <Pagination flag> <Compression flag>
<Task name> <Task entry point> <Task exit point>

<TACT Report Header> is a title which appears at the top of every TACT Report page. The title can be up to 40 characters in length.

<Clock skip> is the number of clocks to skip between records in the tact report.

<Pagination flag> is set to 1 if pagination of the TACT report is desired (line printer), 0 if pagination is not desired (terminal).

<Compression flag> is set to 1 if tact Report compression is desired, 0 if multiple identical records are desired.

<Task name> is a 6 character identifier for the task, to be printed on the TACT report.

<Task entry point> is an address or label defining the starting point of the task.

<Task exit point> is an address or label defining the ending point of the task.

The last record can be repeated up to 30 times so that up to 30 tasks can be defined at one time. A task can begin on the same point that another task ends, but tasks can not overlap in memory.

```
>      1      8X8 SPARSE
>      2      100 0
>      3      FAC1          B1        B2
>      4      JOIN1         B14       B15
>      5      MUL2          B31       B32
>      7      JOIN4         $160      B18
>      8      JOIN5         B16       B17
>      9      SOL1          B7        B8
>     10      MUL1          B4        B5
>     11      MUL           365A      421A
>     12      SOL           270A      340A
>     13      FAC           425A      500C
#End of file
```

Listing of Sample Task Definition File

APPENDIX L

Example Use of Simulator and Cross Assembler

The following pages show a sample terminal session in which the Cross Assembler and Simulator package is used to assemble and execute a simple CAL code using a Fortran driver.

The first part of the Fortran driver is the common block containing the simulated CRAY memory (see section 2.3.2). The next portion initializes the simulator and loads the cross assembled object module. Next, the values to be squared are loaded into the simulator memory at address 200 octal for a vector length of 100 octal (MEM array subscripts 129 through 192). After the object code and operands have been loaded, all that remains is to run the simulation and retrieve the results from simulator memory, which the next two sections of the driver perform. The results, of course, are the squares of the first 64 integers, as we expected.

```
#$LIST FTN.TEST

      1      C
      2      C
      3      C     FORTRAN DRIVER FOR VECTOR SQUARE
      4      C
      5      C
      6      C-----------------------------------------------------------
      7      C
      8      C
      9      C   COMMON BLOCK FOR CRAY-1 MEMORY.
     10      C
     11            DOUBLE PRECISION MEM
     12            COMMON /MSIZE/  MEMSIZ
     13            INTEGER*2                                  HMEM(1)
     14            INTEGER                        IMEM(2,4096)
     15            COMMON /MEMORY/ MEM(4096)
     16            EQUIVALENCE    (MEM(1),    IMEM(1,1),    HMEM(1))
     17      C
     18      C
     19      C ... INITIALIZE
     20            CALL CRAY1('INIT;LOAD CAL.O;RETURN!',.TRUE.)
     21      C
     22      C ... SET UP VECTOR TO SQUARE
     23            DO 10 I=1,64
     24               MEM(128+I) = 1.0D0 * I
     25      10    CONTINUE
     26      C
     27      C ... RUN THE CAL CODE
     28            CALL CRAY1('CH P SQUARE;SET TIM=OFF;RUN;RETURN!',.TRUE.)
     29      C
     30      C ... GET RESULT
     31            WRITE(6,100) (MEM(128+I),I=1,64)
     32      100   FORMAT(' ',4F10.2)
     33      C
     34      C
     35            STOP
     36            END
#$RUN *FTN SCARDS=FTN.TEST SPUNCH=-O
#Execution begins   15:12:06
 No errors in MAIN
#Execution terminated   15:12:07  T=0.039  $0.02
#$LIST CAL.SQUARE

      1                IDENT SQUARE
      2                BASE  O
      3                ABS
      4                ORG   20
      5     SQUARE     =     *
      6                A1    100       SET VECTOR LENGTH TO 64
      7                VL    A1
      8                A0    200       LOAD VECTOR TO SQUARE
      9                V1    ,A0,A0
     10                V2    V1*FV1    SQUARE THE VECTOR
     11                ,A0,A0   V2     STORE THE RESULT
     12                EX
     13                END
```

```
#$RUN SF01:CAL SCARDS=CAL.SQUARE SPUNCH=CAL.O SPRINT=*DUMMY*
#Execution begins    15:12:09
#Execution terminated    15:12:10   T=0.056   $0.04
#$RUN -O+K350:MP.4
#Execution begins    15:12:13
 INIT
 LOAD CAL.O
 RETURN
 CH P SQUARE
 SQUARE    DEFINITION USED FROM IDENT SQUARE
 SET TIM=OFF
 RUN
 EXIT     0 AT       22A CPU =   1
 RETURN
        1.00       4.00       9.00      16.00
       25.00      36.00      49.00      64.00
       81.00     100.00     121.00     144.00
      169.00     196.00     225.00     256.00
      289.00     324.00     361.00     400.00
      441.00     484.00     529.00     576.00
      625.00     676.00     729.00     784.00
      841.00     900.00     961.00    1024.00
     1089.00    1156.00    1225.00    1296.00
     1369.00    1444.00    1521.00    1600.00
     1681.00    1764.00    1849.00    1936.00
     2025.00    2116.00    2209.00    2304.00
     2401.00    2500.00    2601.00    2704.00
     2809.00    2916.00    3025.00    3136.00
     3249.00    3364.00    3481.00    3600.00
     3721.00    3844.00    3969.00    4096.00
#Execution terminated    15:12:16   T=0.055   $0.04
```

END

FILMED

2-84

DTIC