CRAY® COMPUTER SYSTEM

1 S/X-MP ARCHITECTURE DIFFERENCES

STV-0841

RECORD OF REVISION

CRAY RESEARCH, INC.

PUBLICATION NUMBER

Revision	Description
	March, 1983 - Original printing.

TABLE OF CONTENTS

TABLE OF CONTENTS (CONT.)

COURSE: X-MP Difference

GOAL: To help the learner become as familiar with the Cray X-MP product as they are with the Cray-1S.

AUDIENCE: Cray software personnel familiar with the Cray-1S including: Machine instructions and their timing, operating system and FORTRAN.

OBJECTIVES:

At the end of the course the learner is able to:

-- Give an "off the cuff" presentation on the architecture of the Cray X-MP or the differences between it and the Cray-1S to a group of professional programmers.

-- Get a job to run on the Cray X-MP and be able to time any part of the job's execution.

-- List the ways (explaining each) that a single job's CPU time can be improved by running on the X-MP system as compared to the Cray-1S.

-- List the ways (explaining each) that job throughput could be increased by using the X-MP system (both with and without an SSD) as compared with the Cray-1.

-- Anticipate possible problem areas (eg., simultaneous memory reads and writes, vector collapse) in both FORTRAN and CAL code to be run on the X-MP and provide solutions where needed.

-- Advise customers and potential customers on possible program and job configurations to take advantage of multi-processing environment.

-- Write a FORTRAN or CAL program that makes use of the multi-processing capabilities of the X-MP running under COS.

SECTION 1

CRAY X-MP ARCHITECTURE

OBJECTIVE: AT THE END OF THE COURSE THE LEARNER IS ABLE TO GIVE A PRESENTATION ON THE ARCHITECTURE OF THE CRAY X-MP OR THE DIFFERENCES BETWEEN IT AND THE CRAY 1-S TO A GROUP OF PROFESSIONAL PROGRAMMERS.

X-MP PHYSICAL CHARACTERISTICS

SIZE

45 SQUARE FEET FLOOR SPACE FOR MAINFRAME
15 SQUARE FEET FLOOR SPACE FOR I/O SUBSYSTEM

WEIGHT

5.25 TONS MAINFRAME
1.5 TONS I/O SUBSYSTEM

COOLING

ENHANCED REFRIGERANT COOLING
- REDESIGNED COLD BARS
- REDESIGNED POWER SUPPLIES

THE MACHINE RUNS AT THE SAME TEMPERATURE AS THE CRAY-1 , BUT HAS THE CAPABILITY TO TRANSFER MORE HEAT AS REQUIRED.

POWER

400 HZ POWER FROM MOTOR GENERATORS
SAME AS FOR CRAY-1

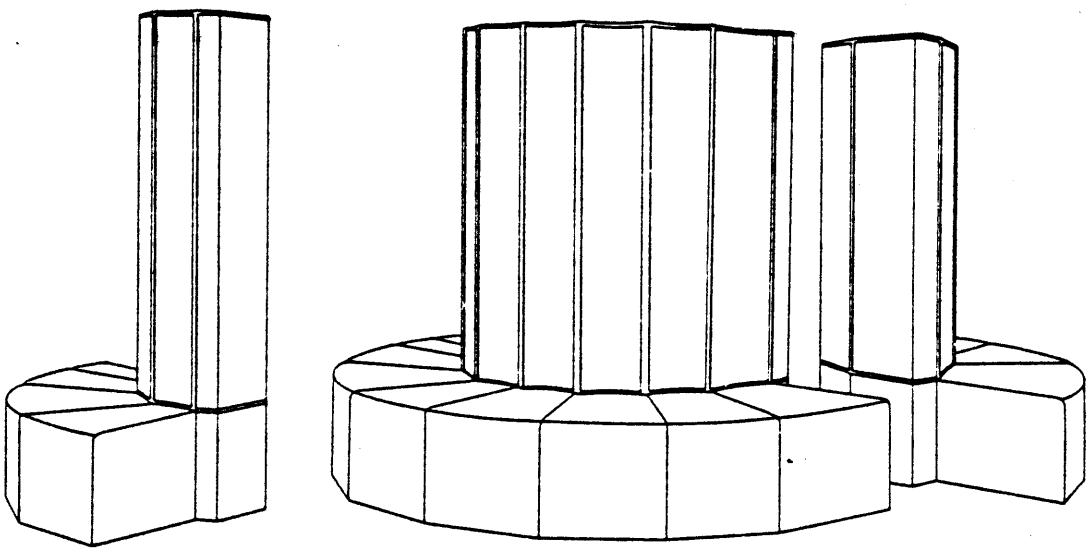
CIRCUITS

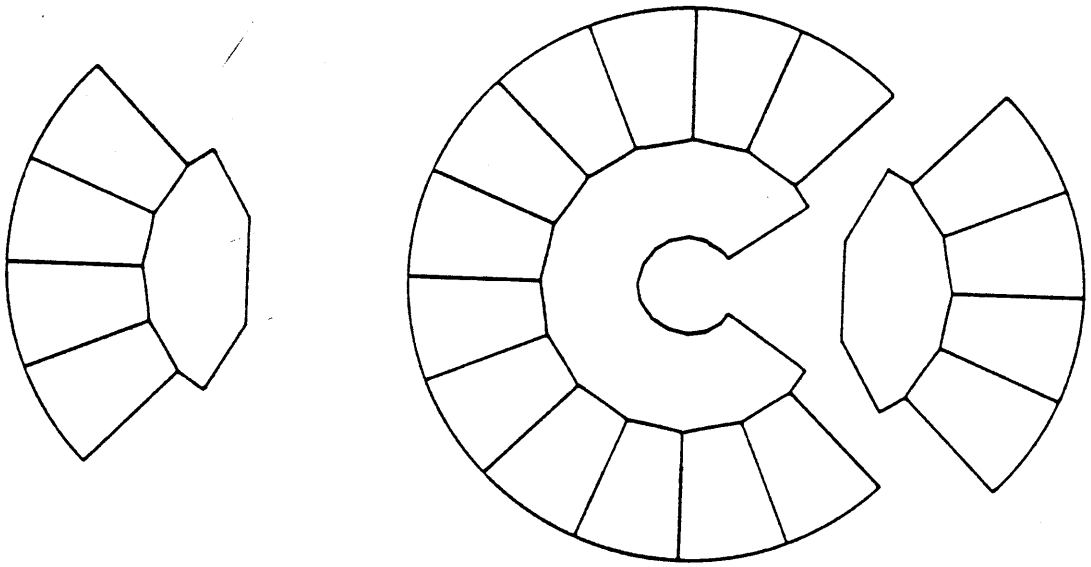
16 GATE-ARRAY INTEGRATED CIRCUITS -- FASTER AND DENSER THAN THOSE USED IN THE CRAY-1.

PCB's

THE PCB'S ARE DOUBLE LAYER AS USED IN THE IOS.

CRAY X-MP PHYSICAL ORGANIZATION





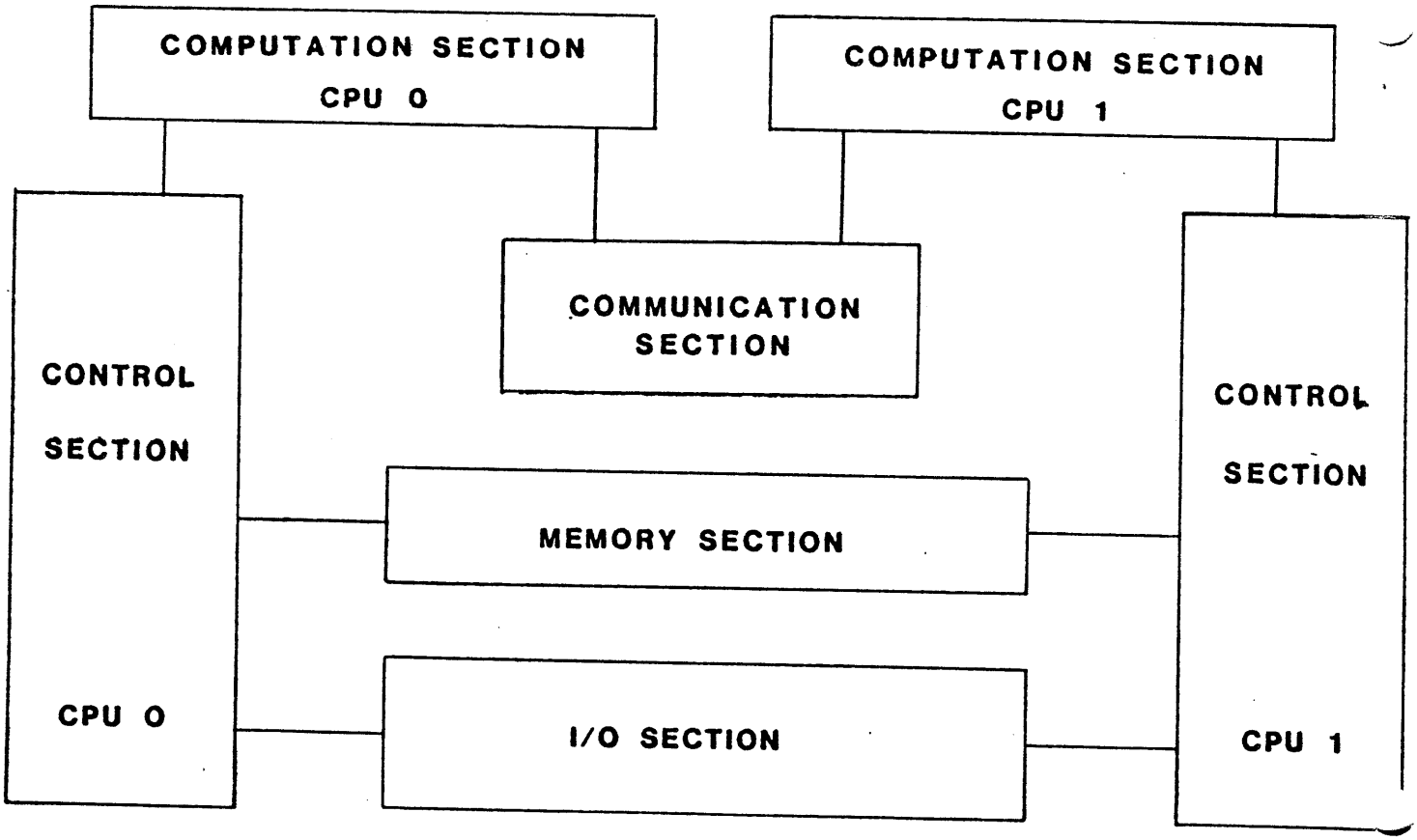
CRAY X-MP BLOCK DIAGRAM

1. 2 IDENTICAL PROCESSORS WITH

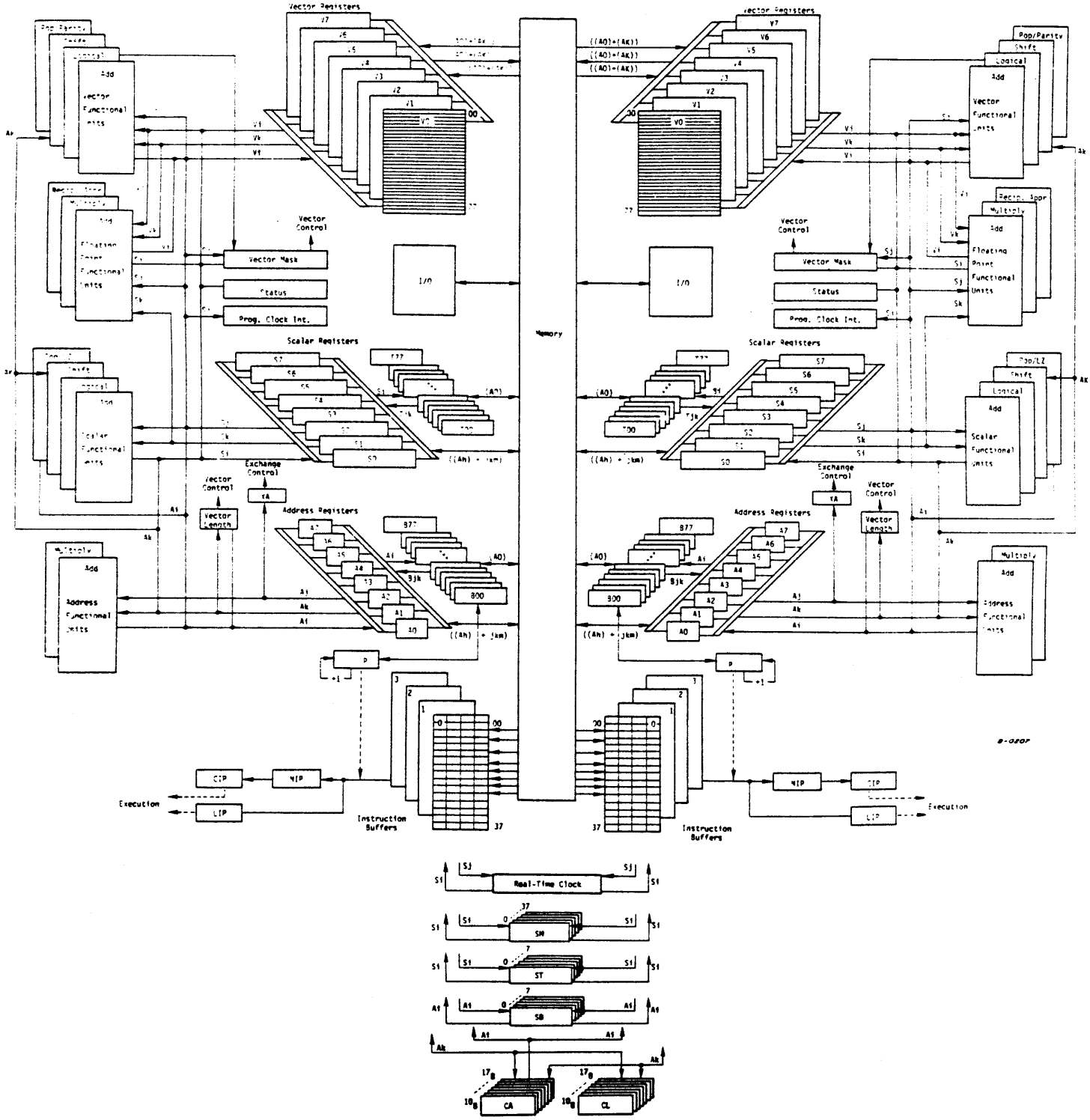
   - SHARED MEMORY
   - A SHARED I/O SECTION
   - INTERPROCESSOR COMMUNICATION REGISTERS
   - INDEPENDENT CONTROL SECTIONS
   - INDEPENDENT COMPUTATION SECTIONS

2. 9.5 NANOSECOND CLOCK PERIOD

3. EACH PROCESSOR IS ARCHITECTURALLY SIMILAR TO THE CRAY 1-S



CRAY X-MP


Vector Registers
Scalar Registers
Address Registers
Memory
I/O
Vector Mask
Status
Prog. Clock Int.
Vector Control
Exchange Control
Vector Length
Instruction Buffers
Execution
Real-Time Clock
SM
ST
SB
CA
CL

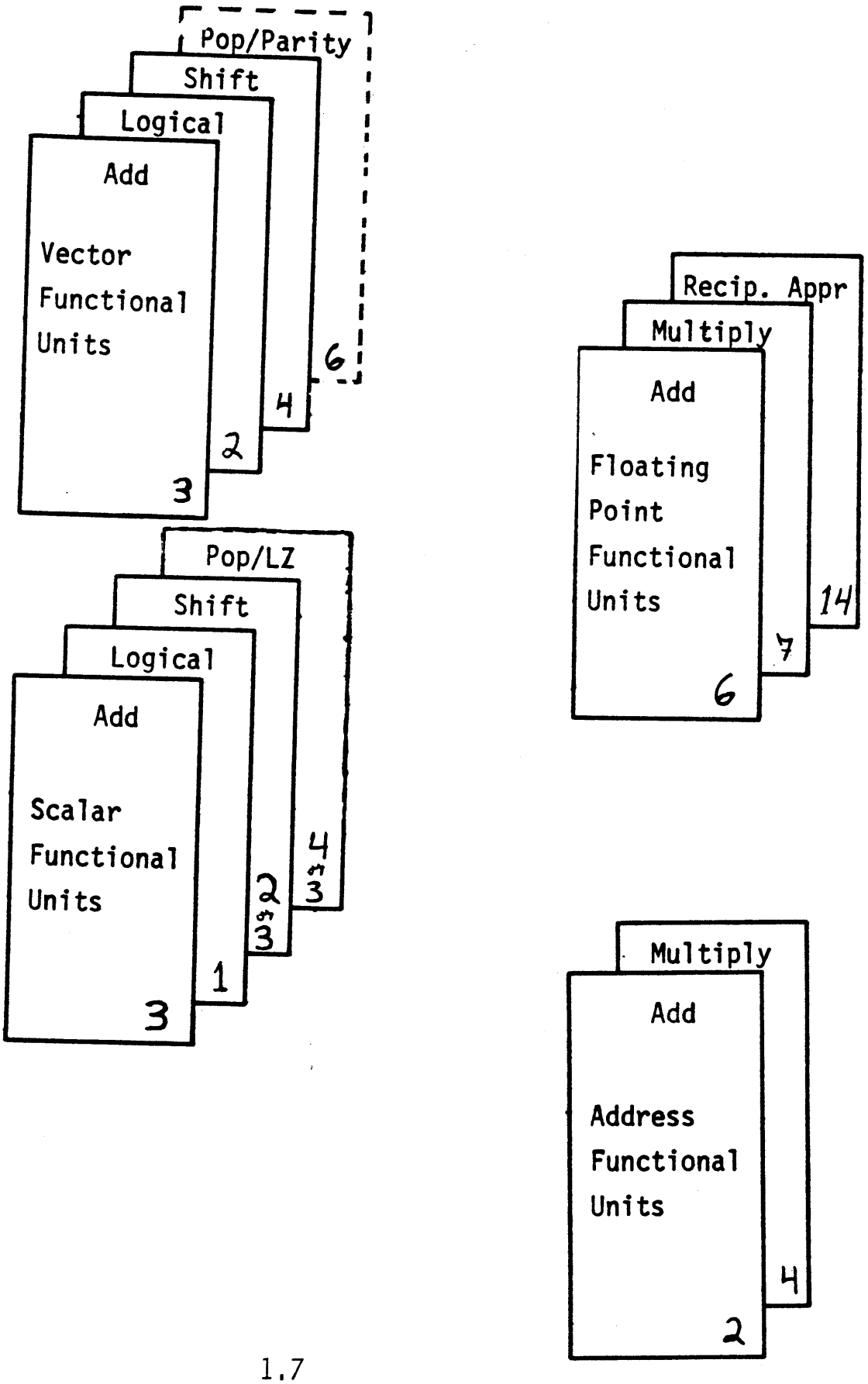
X-MP FUNCTIONAL UNITS

ALL BASIC ARITHMETIC OPERATIONS ARE BIT-COMPATIBLE WITH THE CRAY-1S.

THE NUMBER OF CLOCK PERIODS REQUIRED TO PRODUCE ONE RESULT (THE FUNCTIONAL UNIT TIME) IS THE SAME AS THOSE ON THE CRAY-1S FOR ALL THE FUNCTIONAL UNITS EXCEPT THE ADDRESS MULTIPLY FUNCTIONAL UNIT.

THE 24 BIT MULTIPLE UNITS HAS A FUNCTIONAL UNIT TIME OF 4 CLOCK PERIODS. (IT WAS 6 CP's ON THE 1S.)

FUNCTIONAL UNITS



24 BIT ADDRESS REGISTERS

INTEGER REPRESENTATION IS THE SAME AS IN THE 1-S.

RESULT REGISTERS ARE RESERVED UNTIL THE RESULT ARRIVES.

THERE ARE MULTIPLE PATHS INTO THE A REGISTERS. INSTRUCTIONS ARE NOT HELD BECAUSE OF PATH CONFLICTS.

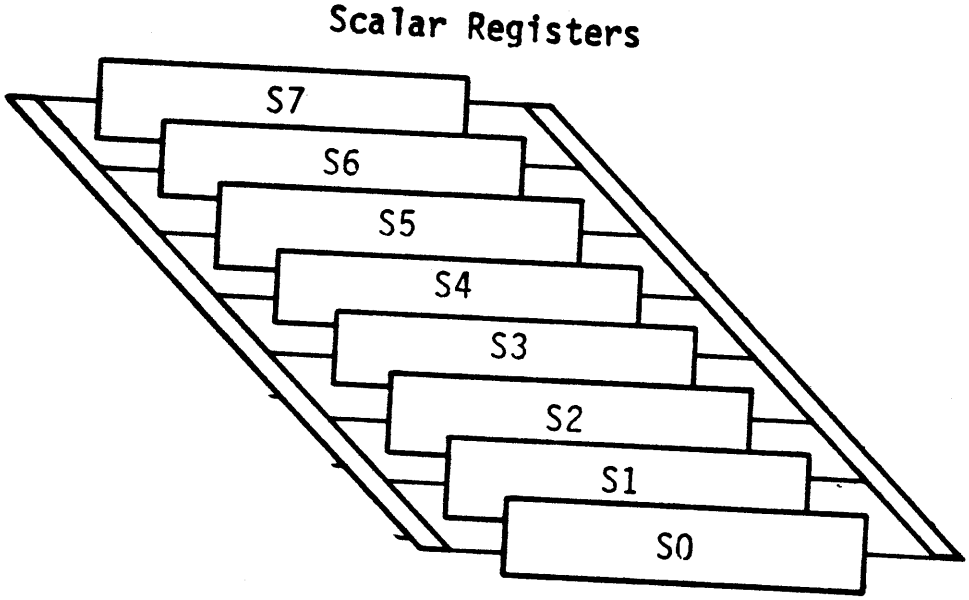
64 BIT SCALAR REGISTERS

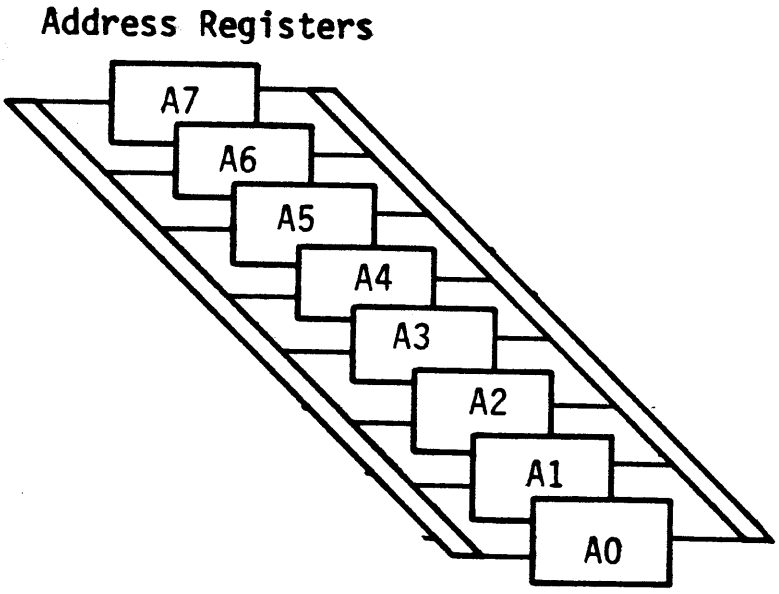
INTERNAL NUMBER REPRESENTATION IS THE SAME AS IN THE 1-S.

RESULT REGISTERS ARE RESERVED UNTIL THE RESULT ARRIVES.

THERE ARE MULTIPLE PATHS INTO THE S REGISTERS. INSTRUCTIONS ARE NOT HELD BECAUSE OF PATH CONFLICTS.

A AND S REGISTERS





AUXILIARY REGISTERS

NUMBER THERE ARE 64 T REGISTERS AND 64 B REGISTERS NUMBERED IN OCTAL 00-77

SIZE: EACH T REGISTER IS 64 BITS WIDE
EACH B REGISTER IS 24 BITS WIDE

TRANSFERS TO AND FROM S AND A REGISTERS IN 1 CP

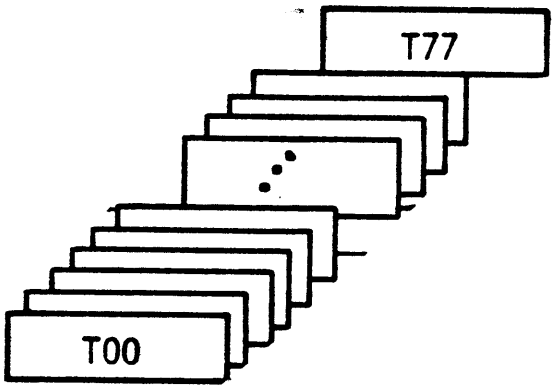
BLOCK TRANSFERS: TO MEMORY: 5 CPS + 1CP/WORD*

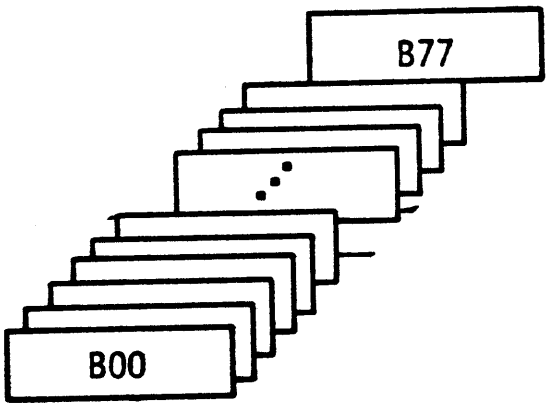
FROM MEMORY: 16 CPS + 1CP/WORD*

*THESE ARE OPTIMAL TIMES, NO MEMORY CONFLICTS

THERE IS NO HOLD ISSUE PLACED ON OTHER INSTRUCTION. ONLY THE B OR T REGISTERS AND ONE PORT TO MEMORY ARE RESERVED.

B AND T REGISTERS





V REGISTERS

8 VECTOR REGISTERS, NUMBERED 0 TO 7

EACH VECTOR REGISTER:

HAS 64 ELEMENTS
EACH ELEMENT IS A 64 BIT CRAY WORD

HAS 2 POINTERS (OR ADDRESS REGISTERS)
ONE RESULT POINTER AND ONE OPERAND POINTER WHEN USED IN THAT ORDER.

THE RESULT POINTER MUST ALWAYS BE AHEAD OF THE OPERAND POINTER. THE OPERAND POINTER MAKES THE REGISTER BUSY.

THIS MEANS THAT A VECTOR REGISTER CAN BE USED BY 2 INSTRUCTIONS AT THE SAME TIME WITHOUT HAVING TO SYNCHRONIZE THEM BY USING CHAIN SLOT TIMES. ALSO
V1 V1+V2 HAS NO SPECIAL MEANING.

HAS MULTIPLE INDEPENDENT INPUT AND OUTPUT PATHS (I.E., THERE ARE NEVER PATH CONFLICTS)

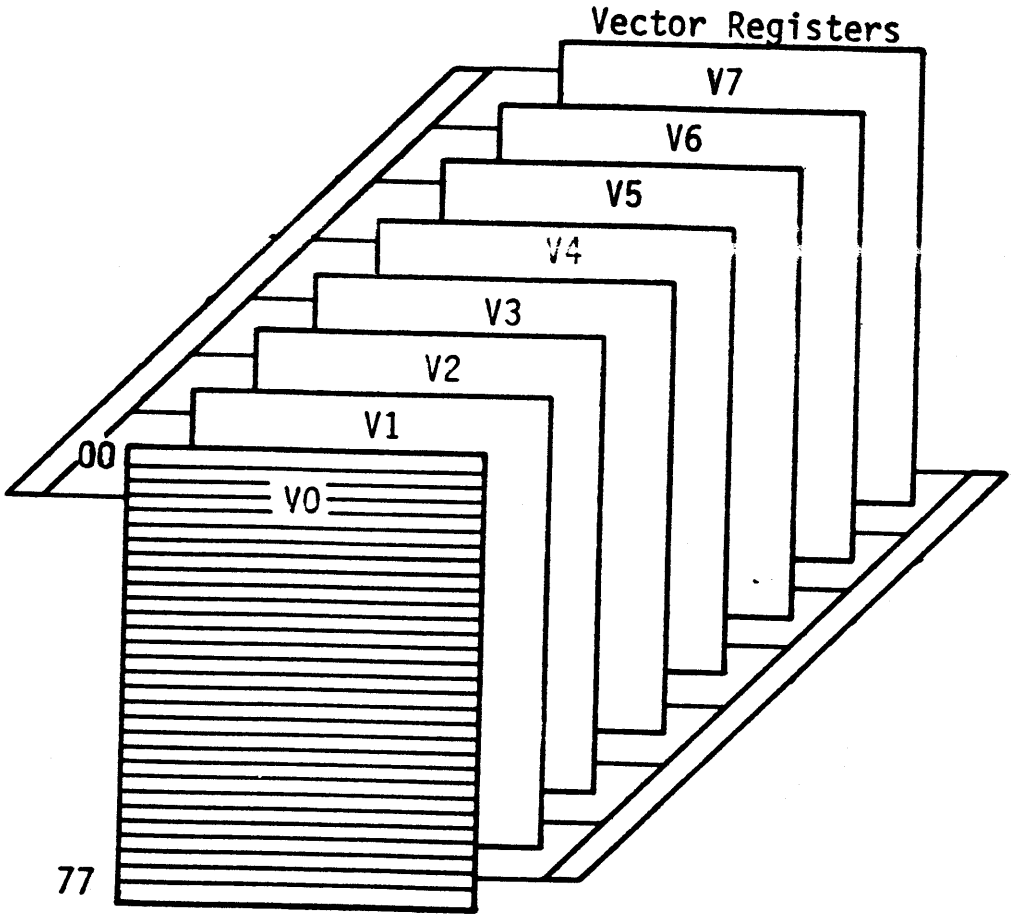
3 MEMORY PORTS ARE AVAILABLE FOR BLOCK TRANSFER OF DATA BETWEEN V REGISTERS AND MEMORY.

PORTS A AND B ARE USED FOR LOADING V REGISTERS

PORT C IS USED FOR STORING TO MEMORY

THE VECTOR LENGTH REGISTER CAN BE DIRECTLY READ AND WRITTEN.

VECTOR REGISTERS



Vector Mask

Vector Length

CENTRAL MEMORY

THE CRAY X-MP PROCESSORS SHARE A SINGLE BIPOLAR CENTRAL MEMORY OF 2M OR 4M 64-BIT WORDS THAT SUPPORTS THE REQUIREMENTS OF LARGE-SCALE APPLICATIONS. MEMORY IS ARRANGED IN 32 BANKS FOR 4 MILLION WORD SYSTEMS AND IN 16 BANKS FOR 2 MILLION WORD SYSTEMS. THESE INTERLEAVED MEMORY BANKS ENABLE EXTREMELY HIGH TRANSFER RATES THROUGH THE I/O SECTION AND PROVIDE LOW READ/WRITE TIMES FOR VECTOR PROCESSING. FINALLY, THE SHORT BANK CYCLE TIME (38 NANOSECONDS) IS WELL-SUITED TO HIGH-PERFORMANCE SCALAR AND VECTOR APPLICATIONS.

A MAJOR FEATURE OF THE CRAY X-MP IS ITS FOUR PARALLEL MEMORY ACCESS PORTS PER PROCESSOR, WHICH INCLUDE TWO PORTS FOR VECTOR READS, ONE FOR VECTOR WRITES, AND ONE FOR I/O. THIS NOTABLE HARDWARE ENHANCEMENT PROVIDES THE CRAY X-MP WITH OVER EIGHT TIMES THE MEMORY BANDWIDTH OF THE CRAY-1.

ALL 8 PORTS ARE USED IN THE INSTRUCTION FETCH FOR EITHER PROCESSOR. A FETCH STOPS NEW REFERENCES AND GOES WHEN BANKS ARE QUIET.

AN EXCHANGE USES ONLY PORTS ASSOCIATED WITH THE PROCESSOR DOING THE EXCHANGE. ALL REFERENCES BY THAT CPU MUST COMPLETE FIRST.

MEMORY IS DIVIDED IN 4 SECTIONS AND 16 OR 32 BANKS (4 OR 8 BANKS PER SECTION).

PORTS COMPETE FOR ACCESS. CONFLICTS CAN BE:

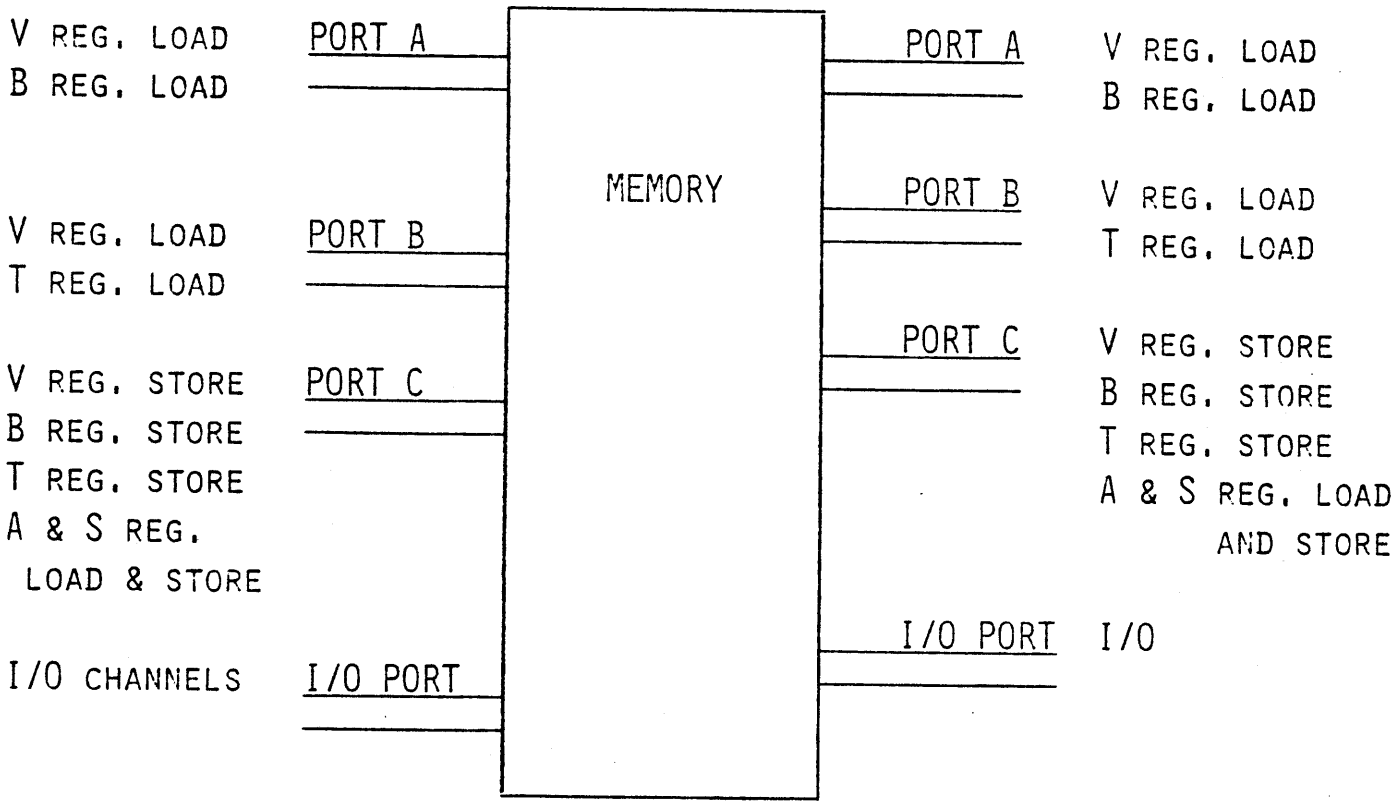
BANK CONFLICTS: ANY 2 PORTS WANT THE SAME BANK WITHIN 4CP's

SECTION CONFLICTS: ANY 2 PORTS OF THE SAME CPU WANT THE SAME SECTION AT THE SAME TIME

CONFLICTS ARE RESOLVED ON AN ELEMENT BY ELEMENT BASIS, I.E., MEMORY ACCESS TIMES ARE NOT DETERMINISTIC.

PAGE 2-3 TO 2-6 OF HR-0032.

X-MP MEMORY



- 2M or 4M words of bipolar IC memory arranged in 16 or 32 banks, respectively
- Shared access from the two CRAY X-MP Processors
- 4 clock periods (38 nanoseconds) bank cycle time
- 4 memory access ports per CPU
- 64 data bits and 8 error correction bits per word
- Single error correction, double error detection (SECDED)

Register to Memory Transfer Rates

Registers	Words per clock period per CPU	Total maximum system transfer rate (Mbits/sec)
B, T, V	3	40,420
A, S	½	6,730
Instruction buffers	8	53,890
I/O	2	13,470

BLOCK LOADS AND STORE

DUE TO THE FACT THAT MEMORY CONFLICTS ON BLOCK LOADS AND STORES ARE RESOLVED ON AN ELEMENT BY ELEMENT BASIS, TIMING OF THESE MEMORY ACCESSES IS NON-DETERMINISTIC.

WAIT FOR A FREE PORT

WHILE THERE ARE MULTIPLE PORTS THERE IS NOT AN INFINITE NUMBER.

INTERRUPTIONS DUE TO FETCHES

A FETCH BY EITHER PROCESSOR USES ALL PORTS AND MAKES MEMORY BUSY.

BANK CONFLICTS

8 PORTS ARE COMPETING FOR MEMORY; BANK CONFLICTS ARE COMPLETELY UNPREDICTABLE.

SECTION CONFLICTS

TWO PORTS FROM THE SAME PROCESSOR CANNOT ACCESS THE SAME SECTION AT THE SAME TIME. MAKING OPTIMAL USE OF THE PORTS INCREASES THE CHANCES FOR SECTION CONFLICTS.

V REGISTER POINTER DELAY

THERE IS NO HOLD ISSUE TO WAIT FOR CHAIN SLOT BUT ACCESS TO THE OPERAND REGISTER IS DELAYED UNTIL IT HAS RECEIVED ITS NEW VALUE. "CHAINING" BEGINS AS WITH THE CRAY-1S.

DELAYS ARE CARRIED THROUGH THE CHIME

DELAYS ENCOUNTERED IN LOADING THE V REGISTERS TO BE USED IN PIPELINED ("CHAINED") OPERATIONS WILL BE CARRIED THROUGH ALL SUBSEQUENT OPERATIONS IN THE PIPE.

```
V0    ,A0,1
V1    S2!V0
,A0,1 V1
```

TIME IN CP's	# OF ELEM ARRIVING IN V0	# OF ELEM LEAVING V0	# OF ELEM ARRIVING IN V1	# OF ELEM LEAVING V1
↓	↑ START UP TIME	⊤ WAIT AFTER ISSUE	↑ Fu+2CP's	↑ WAIT AFTER ISSUE
	↓ 0	↓ 0	↓ 0	↓ 0
	1	1	1	
	3	3	2	
	4	4	3	
	5	5	4	1
	6	6	5	2
	7	7	6	3
	8	8	7	4
	9	9	8	5
			9	6
	10	10	10	7
	11	11	11	8
	12	12	12	9
	13	13	13	10
	14	14	14	11
	15	15	15	12
	16	16	16	13

INSTRUCTION FETCH:

MAY WAIT 0 TO 3 CPS FOR MEMORY TO BE QUIET

16 CP'S FOR THE INSTRUCTION POINTED TO BY THE P REGISTER TO BE IN CIP

+4 CP'S TO LOAD THE WHOLE BUFFER ON A 32 BANK MACHINE

+6 CP'S TO LOAD THE WHOLE BUFFER ON A 16 BANK MACHINE

INSTRUCTION EXECUTION:

PHILOSOPHY IS THE SAME AS IN THE 1-S, I.E., HOLD THE INSTRUCTION IN CIP/LIP UNTIL ALL CONFLICTS ARE RESOLVED.

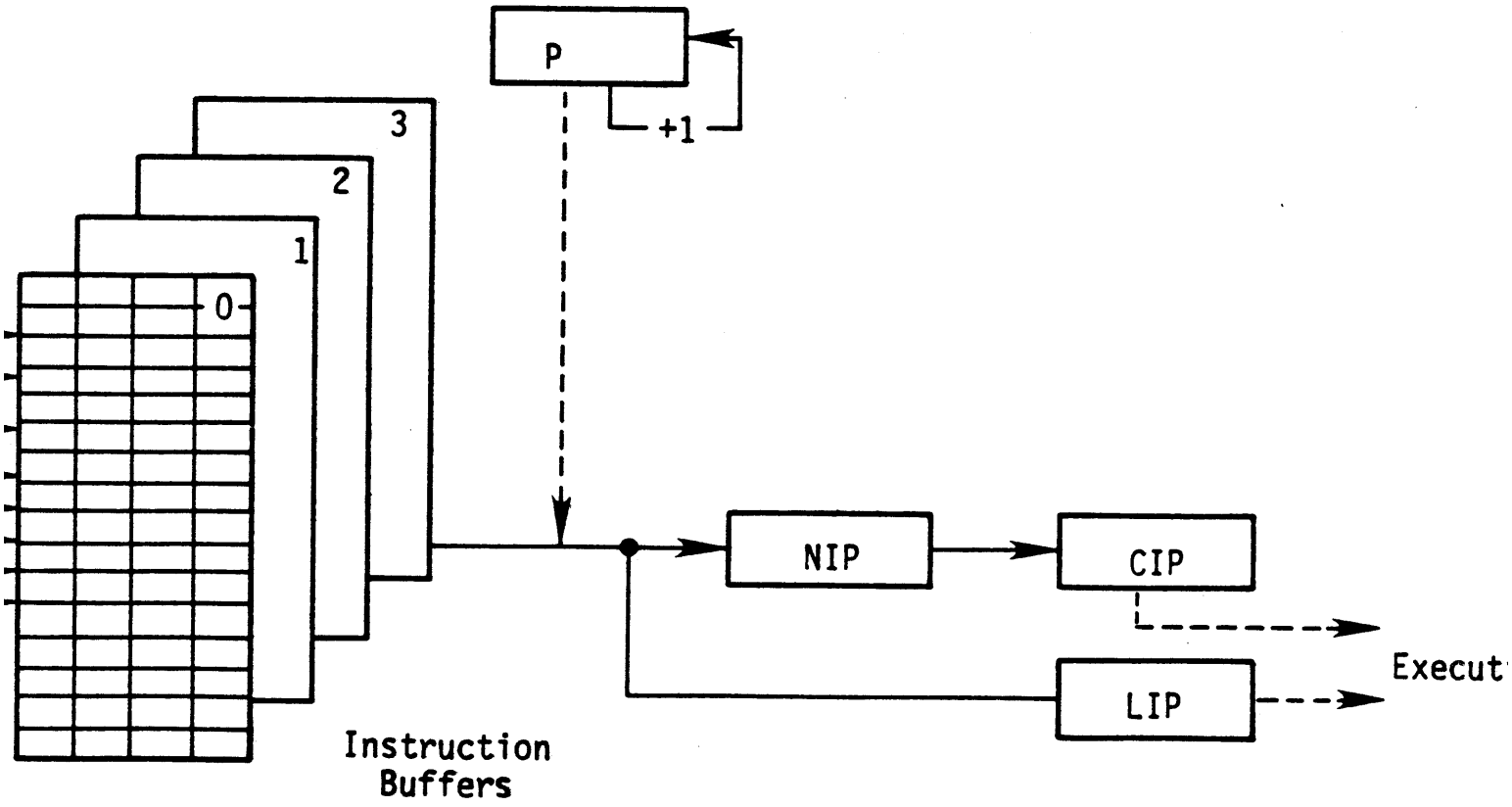
NO PATH CONFLICTS WITH A AND S REGISTERS

2 POINTERS IN THE V REGISTERS (NO MORE CHAIN SLOT TIMES)

A SCALAR MEMORY REFERENCE CONFLICT IS DETECTED IN CP3 OF EXECUTION. IF A CONFLICT OCCURS, ONE MORE SCALAR MEMORY REFERENCE IS ALLOWED TO ISSUE. A THIRD REFERENCE HOLDS ISSUE IF THE CONFLICT STILL EXISTS.

ON A DEADLOCK: AN EXCHANGE TAKES PLACE, NIP & CIP/LIP ARE CLEARED, AND P IS BACKED UP TO POINT TO THE TEST AND SET INSTRUCTION.

CONTROL SECTION



- FOUR INSTRUCTION BUFFERS, EACH HOLDING 128 16-BIT INSTRUCTION PARCELS.
- 128 BASIC INSTRUCTION CODES.
- INSTRUCTION BUFFERS LOADED AT 8 WORDS PER CLOCK PERIOD.
- EXCHANGE MECHANISM
- NORMAL AND INTERPROCESSOR INTERRUPT HANDLING
- SEPARATE PROGRAM AND DATA FIELD PROTECTION IN MEMORY FACILITIATES SHARED CODE AND GREATER PROGRAM PROTECTION.

The I/O Section of the CRAY X-MP, shared by the two CPUs, may be equipped with a variety of high-performance channels for communicating with the mainframe, I/O Subsystem, and a Solid-state Storage Device (SSD).

I/O Channels

- Four 6-Mbyte/sec channels for communication with the mainframe or per connecting front-ends via CRI interfaces (but not NSC adapters).
  - 16 data bits, 3 control bits, and 4 parity bits
- Two 100-Mbyte/sec channels for data transmissions to/from the I/O Subsystem
  - 64 data bits, 3 control bits, and 8 check bits in each direction
- One 1250-Mbyte/sec channel for use with the SSD
  - 128 data bits and 16 check bits in each direction

I/O Subsystem

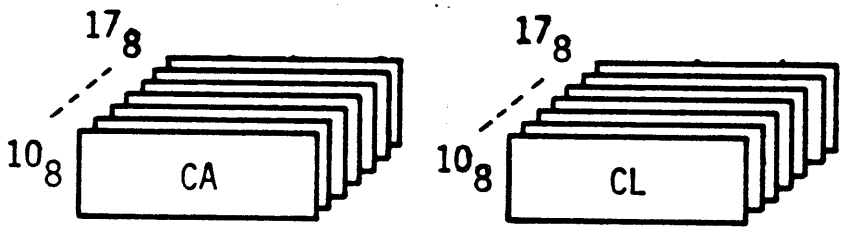
To increase CPU efficiency and encourage parallel I/O processing, no peripheral such as disk units are attached directly to the mainframe. The integral I/O Subsystem is equipped with:

- Two to four I/O Processors
- 12.5 ns clock period
- 8, 32, or 64 Mbytes of Buffer Memory
- Up to 48 600 Mbytes disk storage units
- Optional Block Multiplexer Channels for user supplied tape units
- One to three Cray Research Front-end Interfaces or user-supplied Network System HYPERchannel Adapters
- Operator consoles
- A Peripheral Expander and associated maintenance peripherals

Optional Solid-state Storage Device (SSD)

The SSD connects to the mainframe and is available in sizes of 8, 16, or 32 million words.

I/O SECTION



- 4 6-MBYTE/SEC I/O CONTROL CHANNELS
- 2 100-MBYTE/SEC CHANNELS FOR TRANSFERRING DATA BETWEEN THE IOS AND CENTRAL MEMORY
- 1 1250-MBYTE/SEC CHANNEL FOR TRANSFERRING DATA BETWEEN THE SSD AND CENTRAL MEMORY

THE SAME REAL TIME CLOCK IS ACCESSIBLE TO BOTH PROCESSORS. THEIR CLOCK CYCLES ARE SYNCHRONIZED.

THERE ARE 3 CLUSTERS OF SHARED REGISTERS.

AN EXCHANGE PACKAGE IS GIVEN A CLUSTER NUMBER (CLN) BY COS WHEN IT IS ACTIVATED.

A CLUSTER NUMBER OF ZERO (CLN=0) MEANS NO SHARED REGISTERS ARE AVAILABLE TO THE TASK. ALL INSTRUCTIONS REFERRING TO SHARED REGISTERS ARE TREATED AS NO-OPS.

EACH CLUSTER HAS:

32 1 BIT SEMAPHORE (SYNCHRONIZATION) REGISTERS
8 24 BIT SB (SHARED B) REGISTERS
8 64 BIT ST (SHARED T) REGISTERS

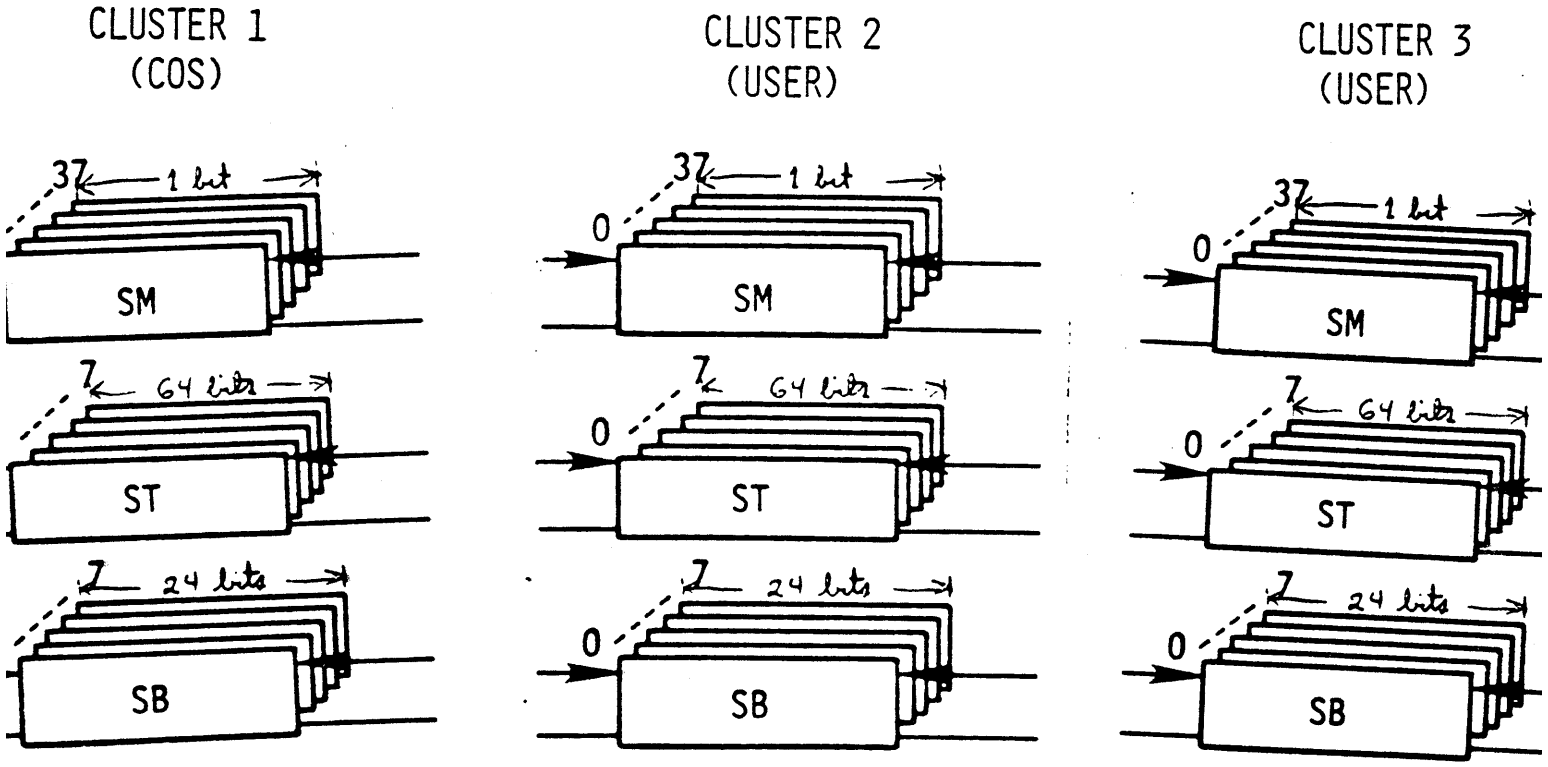
THE EXCHANGE PACKAGES ACTIVE IN BOTH CPU'S (BOTH TASKS) MUST HAVE THE SAME CLUSTER NUMBER IN ORDER TO COMMUNICATE.

SEMAPHORE (SM) REGISTERS CAN BE:
INDIVIDUALLY CLEARED
INDIVIDUALLY SET
INDIVIDUALLY WAITED ON (TEST AND SET)
READ AS A BLOCK
WRITTEN AS A BLOCK

SB AND ST REGISTERS CAN BE:
INDIVIDUALLY READ
INDIVIDUALLY WRITTEN

SHARED REGISTERS

Real-Time Clock



PROCESSOR COORDINATION VIA INTERCOMMUNICATION AND SYNCHRONIZATION REGISTERS

- 3 CLUSTERS OF REGISTERS UNDER COS CONTROL

  8 SHARED ADDRESS
  8 SHARED SCALAR
  32 SYNCHRONIZATION

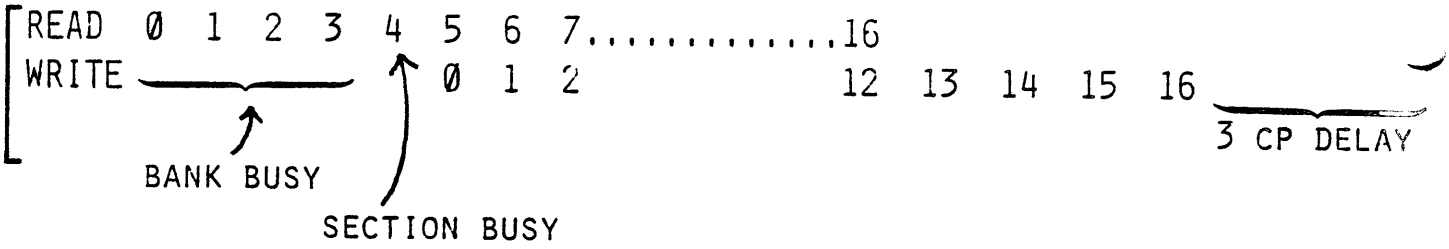
X-MP EXCHANGE MECHANISM

THE EXCHANGE MECHANISM ALLOWS FOR MULTIPROGRAMMING OF EACH OF THE PROCESSORS. WHEN THE CODE ASSOCIATED WITH A TASK IS EXECUTING IN A CPU THE EXCHANGE PACKAGE OF THAT TASK IS ACTIVE. AS IN THE CRAY 1-S, THE EXCHANGE MECHANISM ALLOWS FOR SEVERAL TASKS AND THE OPERATING SYSTEM TO SHARE A PROCESSOR BY ACTIVATING AND DEACTIVATING EXCHANGE PACKAGES.

SOME DIFFERENCES IN THE EXCHANGE MECHANISM ARE:

-- THE TIME TAKEN FOR AN EXCHANGE AND INSUING FETCH.

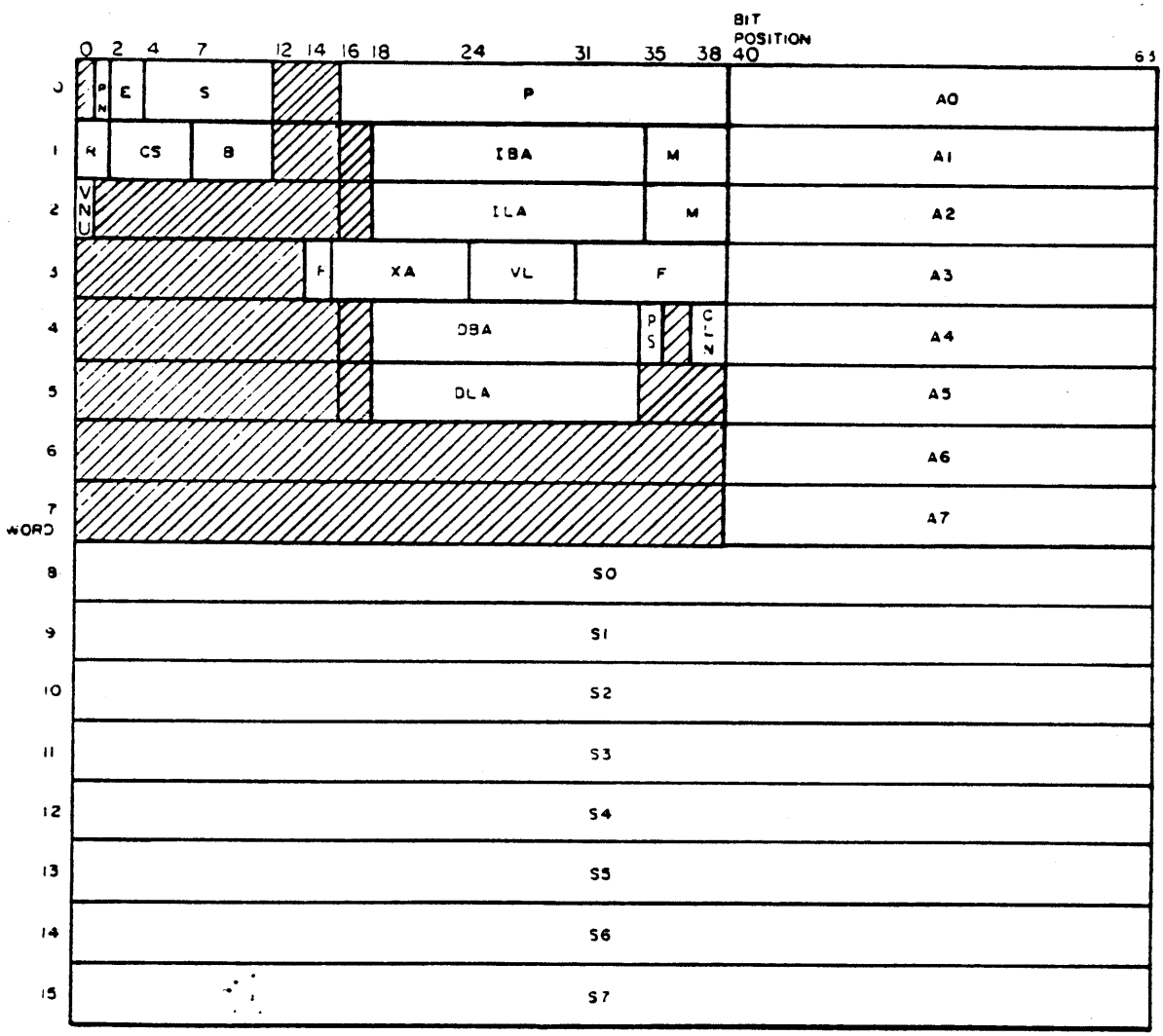
40 CP IN TOTAL: 24 CP's FOR AN EXCHANGE
+ 16 CP's FOR A FETCH



-- THE EXCHANGE PACKAGE HOLDS DIFFERENT INFORMATION

-- MORE INFORMATION IS AVAILABLE TO A TASK BY READING THE STATUS REGISTER

EXCHANGE PACKAGE REGISTERS



P

Status

Vector Length

XA

Si ARj INSTRUCTION
j≠0 GIVES AN OPERAND RANGE ERROR
Si SR0 IS A 073101 INSTRUCTION

CRAY RESEARCH, INC.

X-390

INTEROFFICE MEMORANDUM

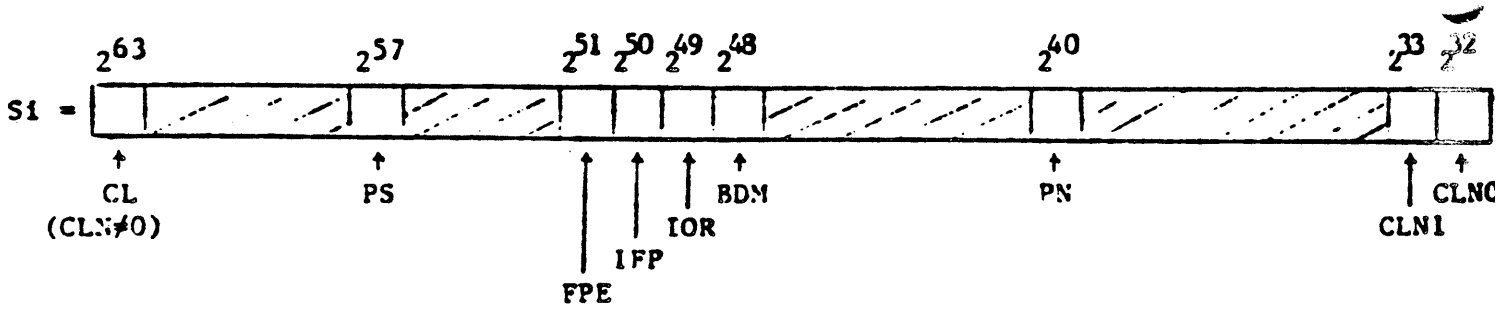
TO: Project File **DATE:** September 16, 1982

FROM: Bob Lutz

SUBJECT: Status Register Change for X-MP

The bit positions will change for the 073101 instruction (Read Status → Si) for X-MP S/N 101 when the field modification is installed.

This instruction will return the following status to the high order bits of Si:



(CL)	Clustered, CLN ≠ 0	→	Si Bit 63
(PS)	Program State	→	Si Bit 57
(FPE)	Floating Point Error Occurred	→	Si Bit 51
(IFP)	Floating Point Interrupt Enabled	→	Si Bit 50
(IOR)	Operand Range Interrupt Enabled	→	Si Bit 49
(BDM)	Bidirectional Memory Enabled	→	Si Bit 48
* (PN)	Processor Number	→	Si Bit 40
* (CLN1)	Cluster Number Bit 1	→	Si Bit 33
* (CLN0)	Cluster Number Bit 0	→	Si Bit 32

*Note: These bit positions return the value of zero if executed in non-monitor mode.

SECTION 2

SINGLE PROCESSOR PROGRAMMING

OBJECTIVE: AT THE END OF THE COURSE THE LEARNER IS ABLE TO LIST THE WAYS (EXPLAINING EACH) THAT A SINGLE JOB'S CPU TIME CAN BE BETTER ON THE X-MP SYSTEM THAN IT IS ON THE CRAY-1S.

SINGLE PROCESSOR PROGRAMMING

MANY PROGRAMS ARE NOT SUITABLE FOR MULTIPROCESSING. THAT IS, THEIR PERFORMANCE WILL NOT BE SIGNIFICANTLY IMPROVED BY BREAKING THE WORK UP INTO TASKS AND HAVING MORE THAN ONE TASK BEING PROCESSED AT A TIME. THESE PROGRAMS WILL RUN UNDER COS IN JUST ONE PROCESSOR. IT DOESN'T MATTER WHICH PROCESSOR IT RUNS IN OR EVEN THAT IT MIGHT RUN IN CPUØ FOR ONE TIME SLICE AND CPU1 FOR THE NEXT. THE TWO PROCESSORS ARE IDENTICAL TO THE PROGRAM.

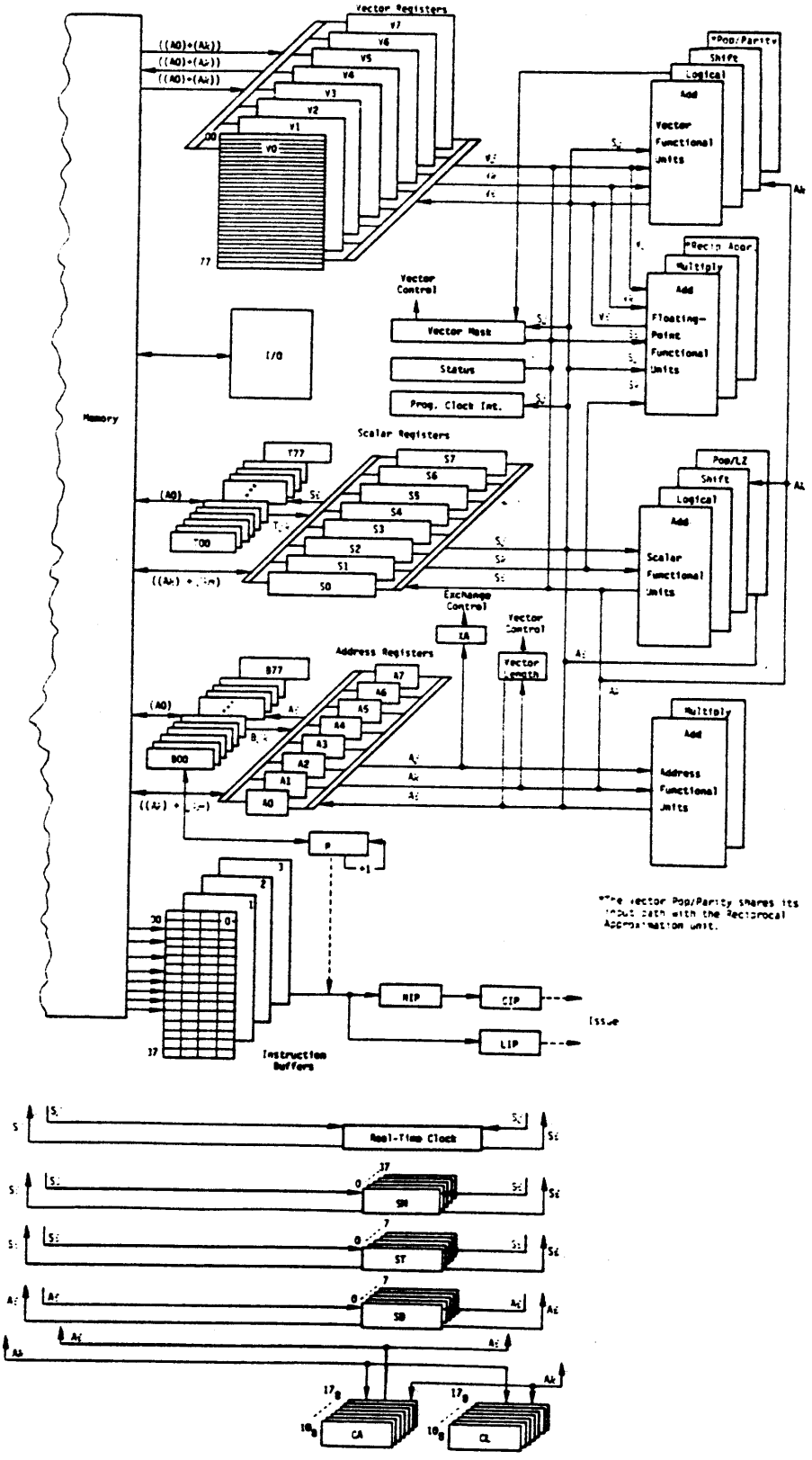
BECAUSE THE ARCHITECTURE OF THE CRAY X-MP IS DIFFERENT FROM THAT OF THE CRAY-1S, THERE WILL BE DIFFERENCES IN HOW TO GET THE MOST OUT OF EACH MACHINE.

DIFFERENCES THAT WILL HAVE THE MOST AFFECT ON PROGRAMMING ARE:

MEMORY ACCESS: 4 PORTS AVAILABLE TO EACH CPU

VECTOR REGISTERS: AUTOMATIC CHAINING, EVEN BACK TO MEMORY

THE SHARED REGISTERS WON'T BE USED AT ALL.


Vector Registers
V7
V6
V5
V4
V3
V2
V1
V0
00
77
((A0)+(Ak))
((A0)+(Aj))
((A0)+(Ai))
*Pop/Parity
Shift
Logical
Add
Vector
Functional
Units
Recip Appr
Multiply
Add
Floating-
Point
Functional
Units
Vector Control
Vector Mask
Status
Prog. Clock Int.
I/O
Memory
Scalar Registers
T77
T00
S7
S6
S5
S4
S3
S2
S1
S0
(A0)
Pop/LZ
Shift
Logical
Add
Scalar
Functional
Units
Exchange Control
XA
Vector Control
Vector Length
Address Registers
B77
B00
A7
A6
A5
A4
A3
A2
A1
A0
Multiply
Add
Address
Functional
Units
P
+1
NIP
CIP
LIP
Issue
Instruction Buffers
*The vector Pop/Parity shares its input path with the Reciprocal Approximation unit.
Real-Time Clock
SM
ST
SB
CA
CL

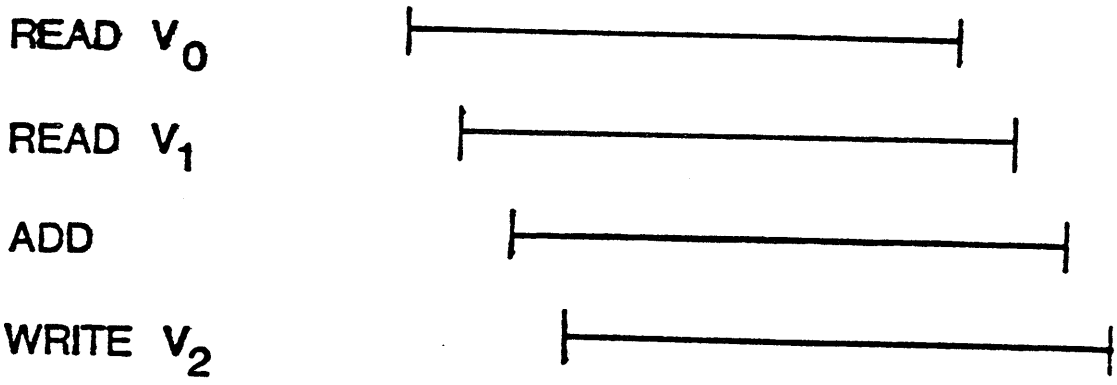
RESULTS ON THE X-MP WILL BE THE SAME AS THOSE ACHIEVED ON THE CRAY-1S.

THE FUNCTIONAL UNITS ARE ARITHMETICALLY IDENTICAL.

THE INTERNAL NUMBER REPRESENTATIONS ARE THE SAME.

IN MOST CASES PROCESSING ORDER IS THE SAME. THE FACT THAT 2 LOADS, CALCULATIONS AND A STORE CAN ALL BE CHAINED WILL NOT AFFECT MOST LOOPS.

$V_0 + V_1 \longrightarrow V_2$



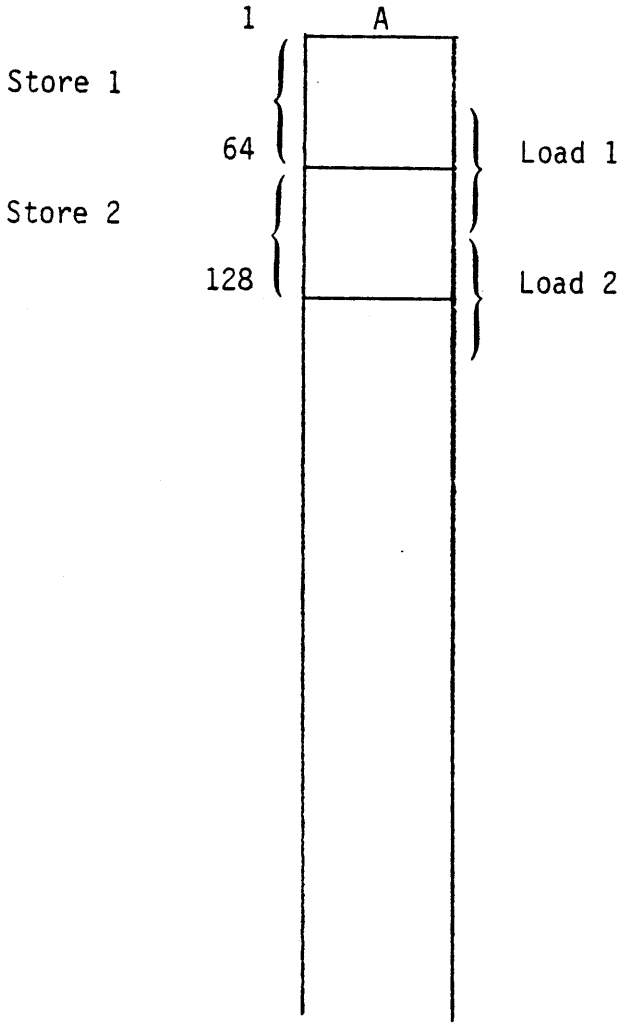
(WILL CHAIN)

USING 3 MEMORY PORTS

SINGLE PROCESSOR PROGRAMMING
(ACCURACY)

Normal vectorizing loops can be run safely (correctly) on the X-MP.

```
      DO 10 I=1, 640
      A(I) = A(I+32)+B(I)
10    CONTINUE
```

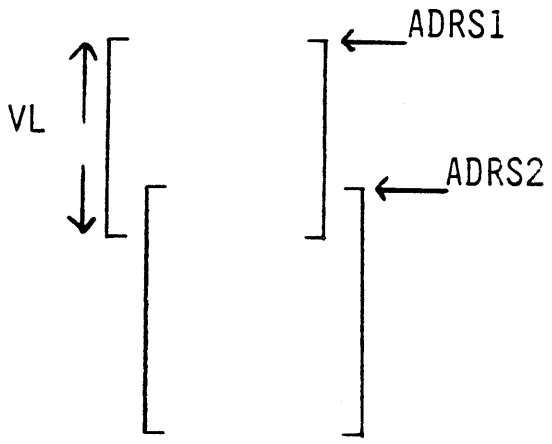


THE CFT COMPILER HAS BUILT IN PROTECTIONS AGAINST THE GENERATION OF WRONG RESULTS DUE TO CHANGE OF PROCESSING ORDER BROUGHT ABOUT BY VECTORIZATION. CARE SHOULD BE TAKEN IN DISABLING THESE PROTECTIONS BY THE USE OF COMPILER DIRECTIVES.

THE CAL PROGRAMMER SHOULD BE AWARE THAT VECTOR LOAD AND STORE OPERATION TIMING IS INDETERMINATE AND SHOULD CHECK FOR POSSIBLE DIFFICULTIES.

EXAMPLE: READ AFTER WRITE ERROR

```
UP    =     *
      .
      .
      .
   A0  ADRS1
   ,A0,1 V7          INSERT CMR (OR A SCALAR MEMORY ACCESS)
      .              HERE AND
      .
      .
   A0  ADRS2         HERE
   V4  ,A0,1
      .
      .
      .
  JSP     UP
```



SINGLE PROCESSOR PROGRAMMING
(ACCURACY)

A new possibility for error!

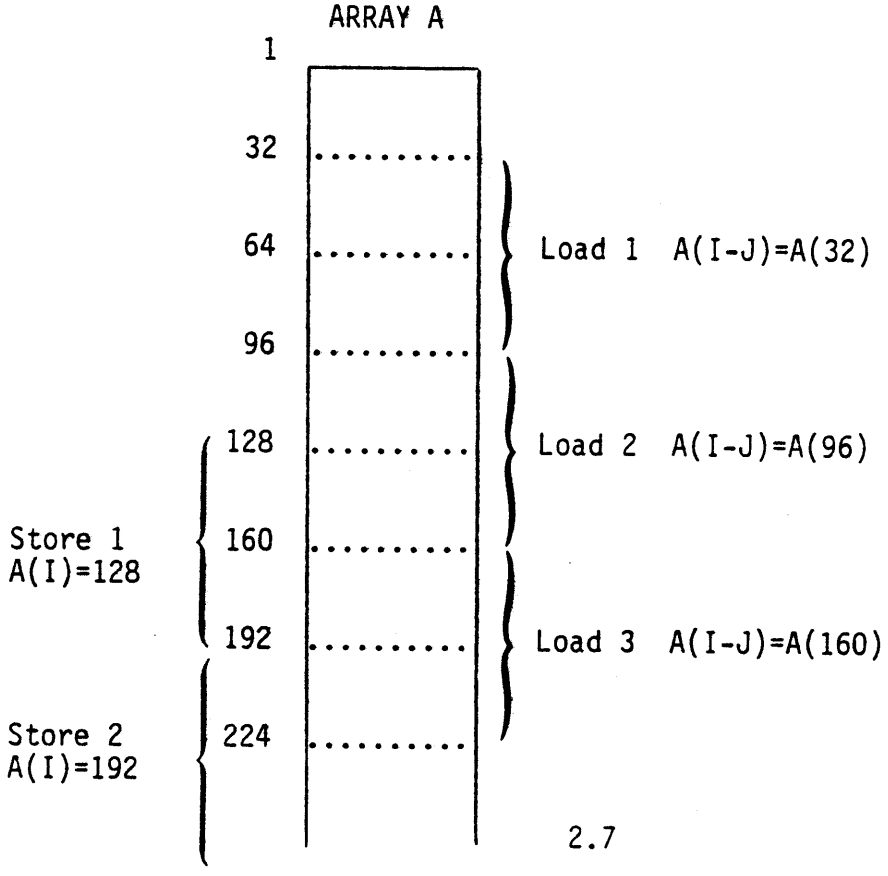
Multiple ports to memory for block loads and stores means overlapping areas of memory can be read from and written to at the same time.

Read after write error
(before the memory area is written to with new values a read can begin that will get old, incorrect values).

for $64 \leqq J < 128$

```
CDIR$ IVDEP
      DO 20 I=128,640

      A(I) = A(I-J)*B(I)

20 CONTINUE
```

example: J=96



TO OPTIMIZE SINGLE PROCESSOR PERFORMANCE ON THE CRAY X-MP, WE MUST TAKE ADVANTAGE OF THE CHANGES MADE IN THE ARCHITECTURE.

SINGLE PROCESSOR PROGRAMMING
(SPEED)

GENERAL OPTIMIZATION STRATEGY

-- MAXIMIZE VECTORIZATION
-- UTILIZE MULTIPORT MEMORY
-- OVERLAP FUNCTIONAL UNITS
   MAKE BEST USE OF CHAINING
-- TAKE MORE ADVANTAGE OF B & T REGISTERS
-- CUT DOWN SCALAR TRAFFIC
-- EMPLOY IOS BUFFER MEMORY AND SSD FOR
   LARGE TEMPORARY FILES
-- EXPLORE NEW ALGORITHMS

MAXIMIZE VECTORIZATION

THE GAIN IN VECTORIZATION IS GREATER ON THE X-MP THAN ON THE CRAY 1-S. FOR THE X-MP, THE CROSSOVER POINT IS SHORTER. THE GAIN IN VECTORIZATION IS BIGGER ON THE X-MP.

THE USER SHOULD BE AWARE OF THE VECTORIZATION TECHNIQUES USED TO IMPROVE PERFORMANCE ON THE CRAY 1-S.

CONTENTS

Speedup of example using this optimization on the (VL = 100, or 1000)

	OPTIMIZATION	SPEEDUP S	SPEEDUP X
3. ARITHMETIC (CONTINUED)			
FORCE PRELOAD SO CHAINING NOT DESTROYED	3-7	1.18	0.76
FORCE PRELOAD OF VECTOR	3-8	1.24	1.03
DON'T UNROLL A LOOP	3-9	1.00	0.97
DON'T USE HORNER'S RULE TO EVALUATE 3RD DEGREE POLYNOMIALS	3-10	0.98	1.
USE HORNER'S RULE TO EVALUATE 4TH DEGREE POLYNOMIALS	3-11	1.08	1.56
USE HORNER'S RULE TO EVALUATE 5TH DEGREE POLYNOMIALS	3-12	1.17	1.82
USE **2 INSTEAD OF SQRT FOR COMPARISONS	3-13	2.04	4.18
USE SQRT INSTEAD OF EXPONENTIAL OF **0.5	3-14	4.28	4.60
USE SQRT INSTEAD OF EXPONENTIAL	3-15	3.71	4.22
REWRITE LOOP TO ENHANCE VECTORIZATION	3-16	1.64	2.3
UNROLL INNER LOOP OF 2	3-17	96.5	47.
UNROLL INNER LOOP OF 3	3-18	63.3	37.
ENHANCE CHAINING IN AN UNROLLED LOOP	3-19	1.07	1.51
UNROLL OUTER LOOP IN INNER LOOP	3-20	1.16	1.56
UNROLL SMALL NESTED LOOPS	3-21a	7.6	3.1
	3-21b	11.4	6.0
4. INTEGER, REAL AND DOUBLE PRECISON			
MATCH ARRAY TYPES FOR INTEGER ADDITION AND SUBTRACTION	4-1	1.7	2.5
FLOAT INTEGER MULTIPLICANDS	4-3	2.8	3.0
AVOID INTEGER MULTIPLICATION	4-4	4.0	8.2
AVOID INTEGER DIVISION	4-5	20.9	36.0
USE SYMBOLIC INTEGER CONSTANTS INSTEAD OF VARIABLES	4-6	2.00	1.95
USE EXPLICIT INTEGER CONSTANTS INSTEAD OF VARIABLES	4-7	2.00	1.95
AVOID DOUBLE PRECISION ADDITION	4-8	66.5	70.8
AVOID DOUBLE PRECISION SUBTRACTION	4-9	69.2	73.3
AVOID DOUBLE PRECISION MULTIPLICATION	4-10	89.7	41.
AVOID DOUBLE PRECISION DIVISION	4-11	136.2	58.
AVOID DOUBLE PRECISION EXPONENTIATION	4-12	324.1	321.0
5. IF			
USE CVMG TO REMOVE IF	5-1	11.1	20.7
MOVE INVARIANT IF OUT OF LOOP	5-3	25.3	71.4
REMOVE IF ON CII	5-5	33.5	93.4
RESTRUCTURE TO ELIMINATE IF	5-6	15.7	31.4
ELIMINATE IF STATEMENTS USING AMAX	5-7	13.3	26.9
MODIFY IF LOOP WITH TEMPORARY ARRAY	5-8	1.14	1.16
6. MEMORY			
GATHER FROM MEMORY	6-1	1.15	1.18
SCATTER TO MEMORY	6-3	1.13	1.15
CHANGE DIMENSION TO ELIMINATE MEMORY CONFLICTS	6-5	3.23	2.58
ELIMINATE MEMORY CONFLICTS BY SWITCHING LOOPS	6-6	1.51	1.02

OPTIMIZATION SPEEDUP

		S	X
7. $SCILIB			
USE $SCILIB SSUM	7-1	2.64	1.33
USE $SCILIB FUNCTION ISMIN	7-3	12.7	14.8
USE $SCILIB FUNCTION ISMAX	7-4	9.3	11.3
$SCILIB SAXPY	7-5	-ERROR-	
USE $SCILIB SDOT	7-6	2.44	1.85
SWITCH LOOPS FOR DOT PRODUCT.	7-7	2.39	2.18
USE $SCILIB SDOT IN NESTED LOOPS	7-8	2.42	1.82
RESTRUCTURE MATRIX MULTIPLY FOR FULL VECTORIZATION	7-9	1.65	1.88
USE $SCILIB MATRIX MULTIPLY	7-10	8.06	5.75
GATHER/SCATTER	7-11	1.24	2.51
8. SUBROUTINES AND FUNCTIONS			
PASS ARGUMENTS TO SUBROUTINES IN COMMON BLOCKS	8-1	1.20	1.30
SUBSTITUTE STATEMENT FUNCTIONS FOR FUNCTION SUBPROGRAMS	8-3	36.4	89.7
PULL SUBROUTINE IN CALLING ROUTINE	8-5	35.5	90.4
PUSH ENTIRE DO LOOP INTO SUBROUTINE	8-7	34.5	81.7
9. I/O			
DO NOT USE IMPLIED DO IN FORMATTED I/O	9-1	1.09	1.09
DO NOT USE IMPLIED DO IN UNFORMATTED I/O	9-3	3.58	3.91
USE UNFORMATTED I/O INSTEAD OF FORMATTED I/O IF POSSIBLE	9-4	132.	147.
USE BUFFERED I/O INSTEAD OF UNFORMATTED I/O	9-5	2.44	1.89

THE CRAY X-MP HAS TWO ELEMENT POINTER IN EACH VECTOR REGISTER (THE CRAY-1S HAS ONE). THIS PROVIDES A FLEXIBLE HARDWARE CHAINING MECHANISM FOR VECTOR PROCESSING. THIS FEATURE ENABLES A RESULT VECTOR TO BE USED AT ANY TIME AS AN OPERAND IN A SUCCEEDING OPERATION. VECTOR REGISTERS ARE BUSY WHEN BOTH POINTERS ARE IN USE. ALSO, VECTOR CHAINING TO MEMORY AS WELL AS FROM MEMORY IS NOW POSSIBLE.

I/O TRANSFERS OCCUR AT A MAXIMUM OF 2-WORD-PER-CLOCK-PERIOD RATE, CONCURRENT WITH CPU MEMORY ACTIVITIES.

VECTOR OPERATIONS WILL RUN MORE EFFICIENTLY ON THE X-MP. FOUR OR MORE OPERATIONS, (EG. 2 READS, ADD AND WRITE) ALL PROCEED IN PARALLEL, OVERLAPPED AND CHAINED TOGETHER. RESULTS CAN BE GENERATED UP TO 4 TIMES FASTER THAN ON CRAY-1 (AT 90 MFLOPS COMPARED WITH 22 MFLOPS).

CONSIDER THE VECTOR OPERATION:

$$C(I) = A(I) + S * B(I)$$

WHERE S IS A SCALAR, A AND B ARE TWO INPUT VECTORS, AND C IS THE OUTPUT VECTOR. THE CRAY X-MP'S MULTIPLE MEMORY ACCESS PORTS PERMITS TWO OPERANDS TO BE READ AND ONE TO BE WRITTEN SIMULTANEOUSLY. IN GENERAL, THE CRAY X-MP ENABLES MEMORY BLOCK TRANSFERS TO THE B,T, AND V REGISTERS IN PARALLEL WITH VECTOR ARITHMETIC OPERATIONS AND I/O TRANSFERS.

UTILIZE MULTIPORT MEMORY

THE INCREASED MEMORY BANDWIDTH EASES THE MEMORY TRAFFIC JAM. IT ALSO OPENS UP NEW OPPORTUNITIES FOR HIGHLY EFFICIENT CODING. THE FLEXIBILITY IN PROGRAMMING AND COMPILER CODE GENERATION IS GREATLY IMPROVED.

EXAMPLE (EASE OF PROGRAMMING)

```
     DO 10 I=1,N                          !LOAD S
10      C(I)=A(I)+S*B(I)                  !LOAD B,*,LOAD A,+,STORE C
```

EXAMPLE

```
     DO 10 I=1,N                          !LOAD B, LOAD A,*
10      C(I)=C(I)+A(I)*B(I)               !LOAD C,+,STORE C
```

EXAMPLE

```
     DO 10 I=1,N
10      A(I)=A(I)+A(I)*B(I)               !LOAD B, LOAD A,*
                                          !+,STORE A
```

OVERLAP FUNCTIONAL UNITS

KEEPING THE FUNCTIONAL UNITS BUSY IS THE KEY TO OPTIMIZATION. THE MORE FUNCTIONAL UNITS THAT ARE PRODUCING RESULTS IN ANY GIVEN CP THE HIGHER OUR MFLOP RATE WILL BE OVERALL. THE EASIEST WAY TO KEEP FUNCTIONAL UNITS BUSY IS WITH THE VECTOR REGISTERS.

THE V REGISTERS IN THE CRAY X-MP ARE MORE EFFICIENT SUPPLIERS OF OPERANDS BECAUSE:

THERE IS MORE THAN ONE POINTER INTO A V REGISTER (I.E., CHAINING IS AUTOMATIC)

THERE ARE MULTIPLE PORTS TO MEMORY

CHAINING OF STORES IS POSSIBLE

MAKE THE BEST USE OF CHAINING

CHAINING IS ONE WAY OF OVERLAPPING VECTOR OPERATIONS. IT IS NEXT BEST TO TOTAL OVERLAPPING, BUT IS USED MORE FREQUENTLY. WITH THE HELP OF MULTIPLE PORT, CHAINING BECOMES MORE POWERFUL.

EXAMPLE (HORNER'S RULE)

```
      DO 10 I=1,N
10       A(I)+(B(I)*X**3)+(C(I)*X**2)+(D(I)*X)+E(I)
```

IT IS MUCH BETTER TO DO:

```
      DO 10 I=1,N
10       A(I)=((B(I)*X+C(I))*X+D(I))*X+E(I)
```

```
LOAD B,*,LOAD C,+
*,LOAD D,+,LOAD E
*
*
```

EXAMPLE

```
      DO 10 I=1,N
10       C(I)=A(I)+S*B(I)
```

IT IS BETTER TO UNROLL THE LOOP:

```
      DO 10 I=1,N,2
         C(I)=A(I+1)+S*B(I+1)
10       C(I)=A(I)+S*B(I)
```

EXAMPLE

```
      V3=0
      LOOP N TIMES
      V7=S1*V1+S2*V2+V3+v4
```

THE INNER LOOP SHOULD BE REARRANGED AS

```
      V3=0
      LOOP N TIMES
           FETCH V1
           FETCH V4
           V1=S1*V1
           V5=V3+V4
           FETCH V2
           V2=S2*V2
           V6=V1+V5
           V3=V2+V6
      ENDLOOP
```

TAKE MORE ADVANTAGE OF B & T REGISTERS

NOW THAT BLOCK TRANSFER DOES NOT HOLD ISSUING, IT IS MORE DESIRABLE TO USE B & T REGISTERS TO STORE BLOCKS OF TEMPORARIES AND CONSTANTS. THEY WERE BUILD AS CACHE MEMORY AND THEY SHOULD BE USED AS SUCH.

CUT DOWN SCALAR MEMORY TRAFFIC

THE SCALAR PERFORMANCE IS MORE IMPORTANT ON THE X-MP. MEMORY TRAFFIC CONTRIBUTES A BIG PART OF THE SCALAR OPERATIONS. CFT CAN BE MADE MORE EFFICIENT.

EXAMPLE: SCALAR ITERATIONS

```
      DO 10 I=1,N
10       A(I)=B(I)/A(I-1)
```

PASS THE A'S HOLDING THEM IN AN S-REGISTER INSTEAD OF MEMORY. THE SAVING WILL BE SIGNIFICANT.

EXAMPLE: SCALAR ITERATIONS

```
      DO 10 I=1,N
10       A(I)=A(I-1)+B(I)
```

BESIDES THE PREVIOUS TECHNIQUES, PREFETCH OF B WILL HAVE A BIG EFFECT TOO. THE SAVE IS ABOUT A FACTOR OF 2.3.

EMPLOY IOS BUFFER MEMORY AND SSD FOR LARGE TEMPORARY FILES

FOR 2-D OR 3-D SIMULATIONS OF LARGE SCALE PROBLEMS, INTERMEDIATE DATA NEED NOT BE SAVED ON DISKS. OFTEN USED SYSTEM FILES CAN ALSO TAKE ADVANTAGE OF THE FAST TRANSFER RATE OF BUFFER MEMORY AND SSD.

EXAMPLE (A STRUCTURAL ANALYSIS BENCHMARK CODE ON THE CRAY-1/S)

I/O CONFIGURATION	I/O WAIT SPEEDUP	THROUGHPUT SPEEDUP
DISK ONLY	1	1
1 MW BUFFER MEMORY	1.56	1.26
4 MW BUFFER MEMORY	7.4	1.98
8 MW SSD	13.32	2.12

Marketing Support Newsletter

CRAY-X/MP I/O PERFORMANCE

An Article Submitted by Dave Slowinski 10/82

The following performance data was obtained by running programs on the thirty-two bank CRAY-X/MP prototype. All I/O requests were done with recall and there were not other jobs running.

"Access Time" was measured as the time to read a random disk sector from a 24 MB file. DD-29 access times vary from 8 msec to 50 msec depending on file size.

All times were measured with the CPU real time clock and include all I/O library and system overhead time.

	Access Time	Transfer Rate
'Clean' DD-29 Disk	25000 usec	4 MB/sec
4 MW Buffer Memory	860 usec	40 MB/sec
8 MW SSD	375 usec	250 MB/sec

Fragmentation can significantly reduce disk I/O performance but has little affect on I/O to Buffer Memory or SSD.

The SSD transfer rate is limited by the number of banks and the bank cycle time. I expect a 16MW SSD to be twice as fast and a 32 MW SSD four times as fast at the 8 MW model.

DS

CRAY RESEARCH, INC.

SSD PERFORMANCE WITH CRAY-1S

	DD-29 (CLEAN)	BUFFER MEMORY	SSD
MEAN ACCESS TIME (microseconds)	25000	1240	985
RELATIVE RANDOM I/O	1	20	25
TRANSFER ONE 512-WORD SECTOR (microseconds)	1200	120	40
SUSTAINED TRANSFER RATE (Megabytes/second)	4	34	102
RELATIVE SEQUENTIAL I/O PERFORMANCE	1	10	30

EXPLORE NEW ALGORITHMS

TO TAKE ADVANTAGE OF THE NEW ARCHITECTURE FEATURES OF VECTOR PROCESSORS AND MULTIPROCESSORS, NEW ALGORITHMS TO DEAL WITH OLD PROBLEMS IS ESSENTIAL. THE DEMAND FOR INCREASED COMPUTING POWER CANNOT BE FULFILLED BY RAW HARDWARE OR COMPONENT SPEED ALONE. MORE OFTEN THAN NOT, INNOVATIVE IDEAS CAN BRIDGE THE GAP BETWEEN REALITY AND WILD DREAMS.

EXAMPLE PROFESSOR CALAHAN'S WORK

CRAY X-MP vs CRAY-1S TIMINGS

* **SAXPY**

Vector Length	RATE IN MFLOPS 1/S	X-MP
5	3.3	5.8
10	6.3	11.3
50	21.0	46.4
100	29.0	74.1
250	37.0	117.5
500	41.0	145.0
1000	43.0	164.2
2500	44.0	176.6
5000	44.0	183.1

* **FFT**

Vector Length	RATE IN MFLOPS 1/S	X-MP
8	4.80	9.75
16	10.6	22.7
32	18.9	43.2
64	29.7	68.3
128	38.7	90.7
256	45.3	104.9
512	50.1	113.5
1024	53.3	118.8
2048	55.9	122.8
4096	57.0	125.9
8192	59.6	128.4

Sensitive Information

* **Do not hand out hard copy of narrative for the slide. These performance numbers are listed here for your convenience only!**

SECTION 3

MULTIPROCESSING: BASIC CONCEPTS

OBJECTIVE: AT THE END OF THE COURSE THE LEARNER IS ABLE TO APPLY UNDERSTANDING OF THE BASIC CONCEPTS OF MULTIPROCESSING IN DECIDING WHETHER OR NOT MULTIPROCESSING IS A REASONABLE APPROACH TO TAKE FOR THE SPEEDING UP OF A GIVEN PROGRAM.

MULTIPROGRAMMING

MULTIPROGRAMMING IS A MODE OF OPERATION THAT PROVIDES FOR THE SHARING OF PROCESSOR RESOURCES AMONG MULTIPLE INDEPENDENT SOFTWARE PROCESSES.

THE CRAY 1 OPERATING SYSTEM COS 1.11 IS A MULTIPROGRAMMING OPERATING SYSTEM. THE PROCESSOR RESOURCE IS JUST ONE CPU AND THE SOFTWARE PROCESSES ARE JOBS. THE SHARING IS DONE BY THE JOB SCHEDULER BY ASSIGNING PRIORITIES TO JOBS AND ALLOCATING CPU TIME A SLICE AT A TIME TO DIFFERENT JOBS.

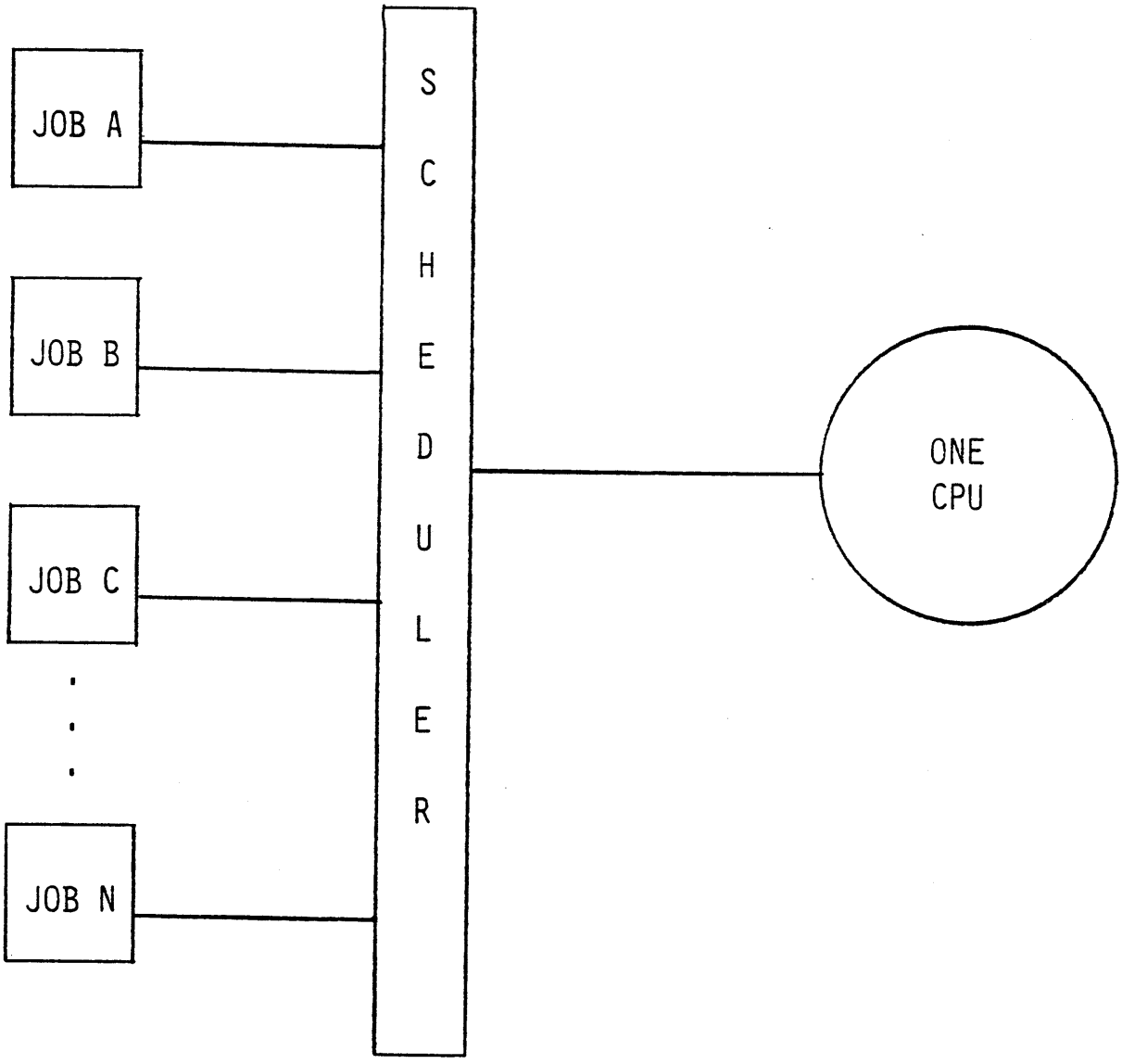
THE PROCESSOR RESOURCE CAN BE MORE THAN ONE CPU. TWO CPU'S COULD BE SHARED BY SEVERAL SOFTWARE PROCESSES. THE SOFTWARE PROCESSES NEED NOT BE JOB; THEY COULD BE JOB STEPS, PROGRAMS, OR EVEN PARTS OF PROGRAMS.

MULTIPROGRAMMING

(MULTIPLE PROGRAMS)

SEVERAL SOFTWARE PROCESSES

PROCESSOR RESOURCES



MULTIPROCESSING

MULTIPROCESSING IS A MODE OF OPERATION THAT PROVIDES FOR PARALLEL PROCESSING BY TWO OR MORE PROCESSORS.

PARALLEL HERE REFERS TO THE MANNER IN WHICH SOFTWARE PROCESSES ARE CONSIDERED. JOBS, PARTS OF JOBS (JOB STEPS), PROGRAMS, EVEN PARTS OF PROGRAMS ARE PROCESSED SIMULTANEOUSLY (OR NEARLY SO) RATHER THAN SEQUENTIALLY OR IN SOME OTHER SPECIAL ORDER.

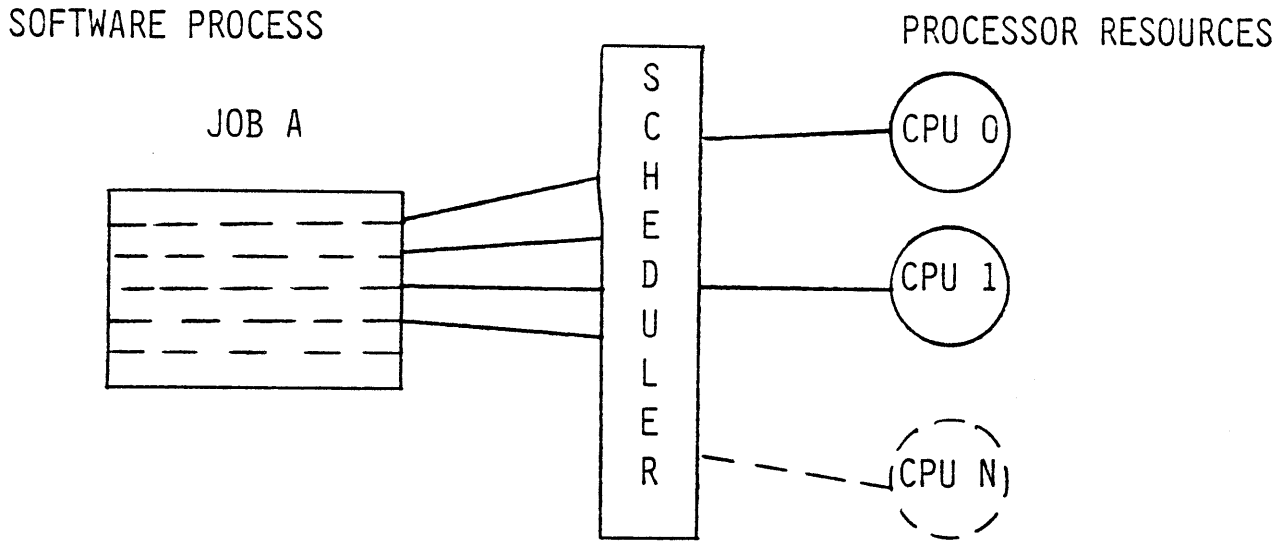
LEVELS OF PARALLELISM CAN BE DEFINED IN TERMS OF THE SOFTWARE PROCESSES THAT CAN BE DONE IN PARALLEL.

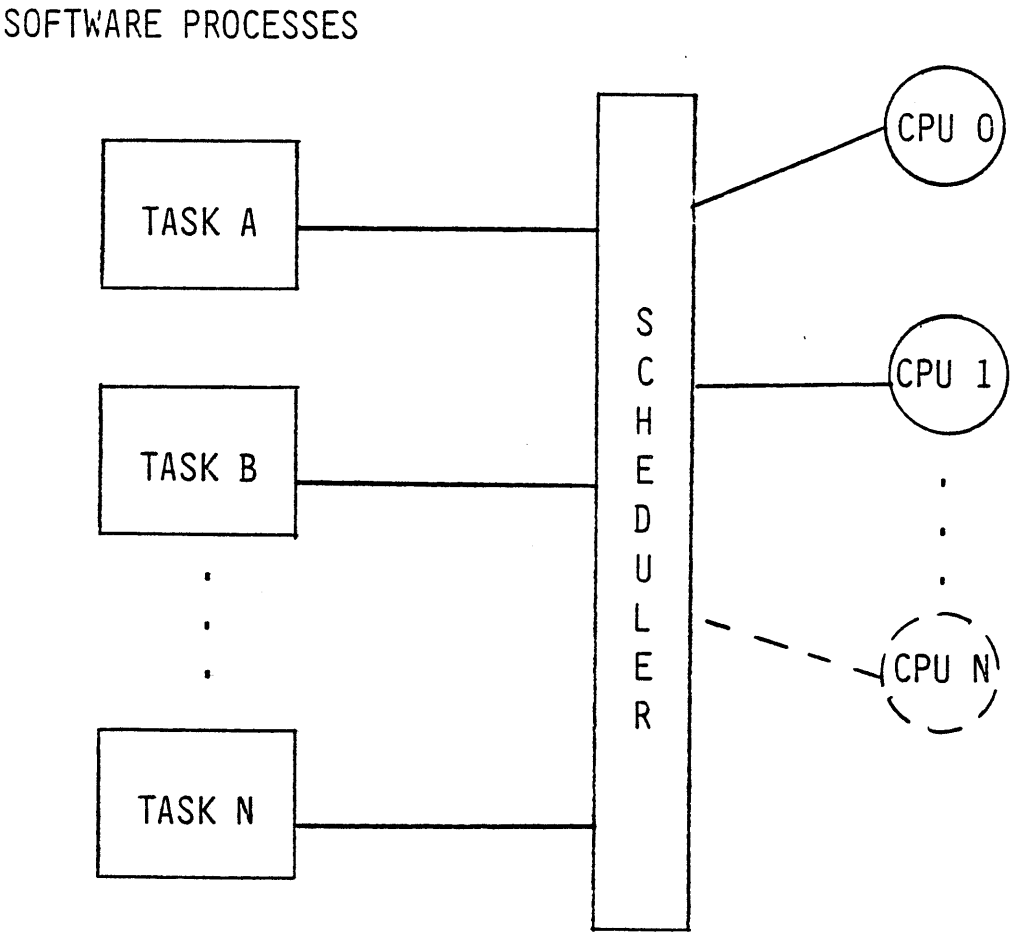
LEVEL 1	JOBS; INDEPENDENT JOBS. EACH HAS A CPU.
LEVEL 2	JOB STEPS; RELATED PARTS OF THE SAME JOB.
LEVEL 3	ROUTINES/SUBROUTINES;
LEVEL 4	LOOPS;
LEVEL 5	STATEMENTS;

THIS DOES NOT IMPLY THAT THE SOFTWARE PROCESSES MUST BE DIFFERENT. THE SAME CODE COULD BE RUN ON 2 PROCESSORS AT THE SAME TIME BUT BE ACTING ON DIFFERENT DATA.

MULTIPROCESSING

(MULTIPLE PROCESSORS)





TASK

A TASK IS A SOFTWARE PROCESS. IT IS A UNIT OF COMPUTATION THAT CAN BE SCHEDULED AND WHOSE INSTRUCTIONS MUST BE PROCESSED IN SEQUENTIAL ORDER.

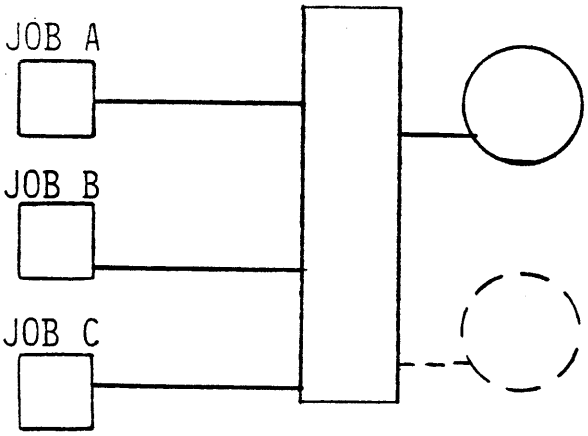
IN A SINGLE PROCESSOR MULTIPROGRAMMING OPERATING SYSTEM LIKE COS 1.11 A JOB IS A TASK.

FOR A JOB TO TAKE ADVANTAGE OF A MULTIPROCESSING OPERATING SYSTEM IT SHOULD INVOLVE MORE THAN ONE TASK. THAT IS, IN ORDER FOR PARTS OF THE JOB TO RUN IN PARALLEL ON MORE THAN ONE PROCESS, THE PARTS MUST BE SCHEDULED SEPARATELY.

WE WILL USE TASK TO MEAN A SUBJOB OR A SUBPROGRAM. A UNIQUELY NAMED PROCESS THAT MAY HAVE CODE AND DATA AREAS IN COMMON WITH (OR EVEN IDENTICAL TO) OTHER TASKS OF THE SAME JOB. WHILE A TASK MAY BE SCHEDULED IT IS NOT NECESSARILY THE SCHEDULING UNIT OF THE OPERATING SYSTEM (I.E., THERE MAY BE OTHER THAN A ONE-TO-ONE MAPPING OF LOGICAL CPU'S ONTO TASKS).

EXAMPLE 1

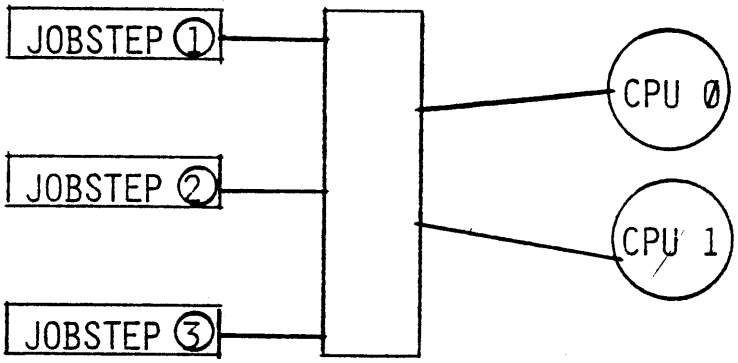
TASK=JOB



1 PROCESSOR (COS 1.11) LEVEL 1

2 PROCESSORS (COS 1.12)

EXAMPLE 2

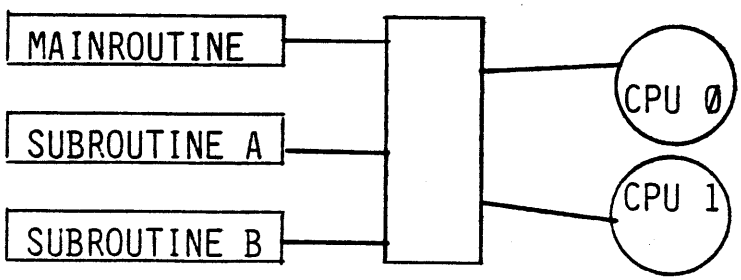


TASK = JOBSTEP

(COS 1.13) LEVEL 2

```
JOB,JN=...
ACCOUNT,AC=...
① CFT,I=DS1,L=Ø    ,B=COMP1
② CAL,I=DS2,L=Ø    ,B=ASM1
③ CFT,I=DS3,L=Ø    ,B=COMP2
REWIND,DN=COMP1:ASM1:COMP2.
```

EXAMPLE 3



TASK = ROUTINE

(COS 1.13) LEVEL 4

MULTITASKING

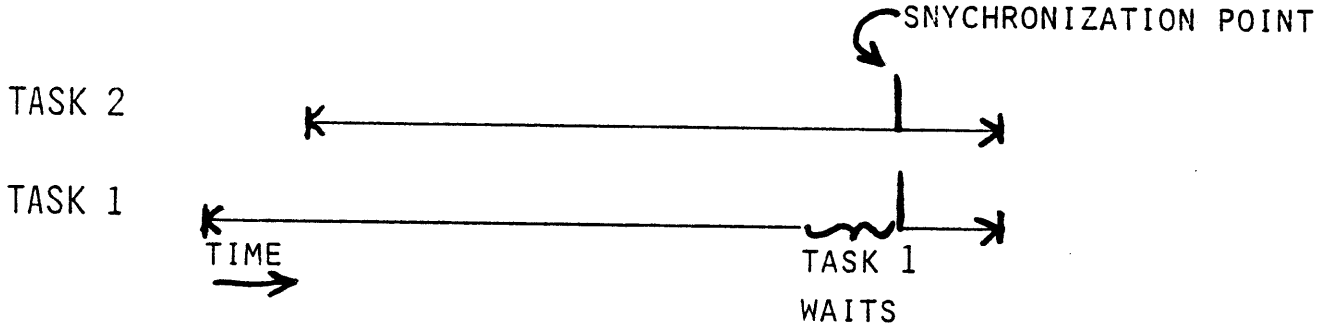
MULTITASKING IS A SPECIAL CASE OF MULTIPROCESSING DEFINING A TASK TO BE A SUBJOB OR SUBPROGRAM; LEVELS 2, 3 AND 4 OF MULTIPROCESSING ARE MULTITASKING MODES OF OPERATION.

IN A MULTITASKING ENVIRONMENT THE TASKS AND DATA STRUCTURE OF A JOB MUST BE SUCH THAT THE TASKS CAN BE RUN IN PARALLEL (CONCURRENTLY OR WITH OPERATIONS OVERLAPPING WITH RESPECT TO TIME). WHILE THERE IS NO GUARANTEE THAT MORE THAN ONE PROCESSOR WILL BE ALLOCATED TO WORK ON THE TASKS OF A GIVEN JOB, THERE IS ALSO NO GUARANTEE OF WHICH OF TWO PARALLEL TASKS WILL FINISH FIRST. MULTITASKING IS NON-DETERMINISTIC WITH RESPECT TO TIME BUT SOFTWARE PROCESSES MUST BE MADE DETERMINISTIC WITH RESPECT TO RESULTS.

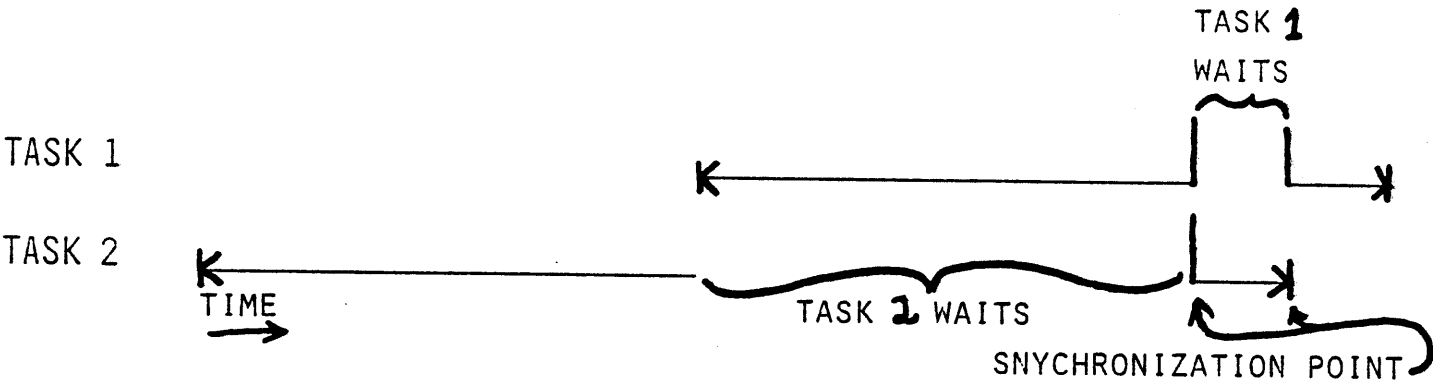
COMMUNICATION BETWEEN PARALLEL TASKS AND PROTECTION OF SHARED DATA MUST BE TAKEN CARE OF BY THE PROGRAMMER.

MULTITASKING

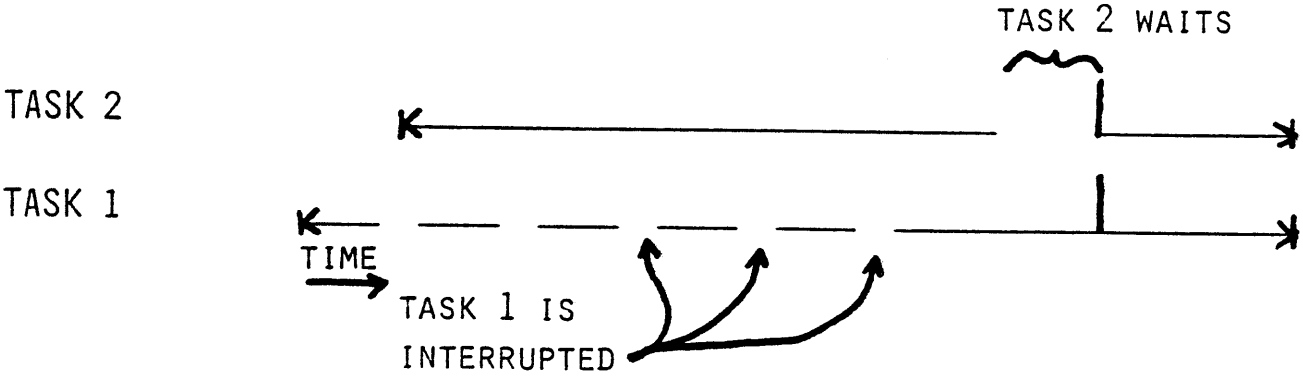
2 PROCESSORS



1 PROCESSOR



2 PROCESSORS



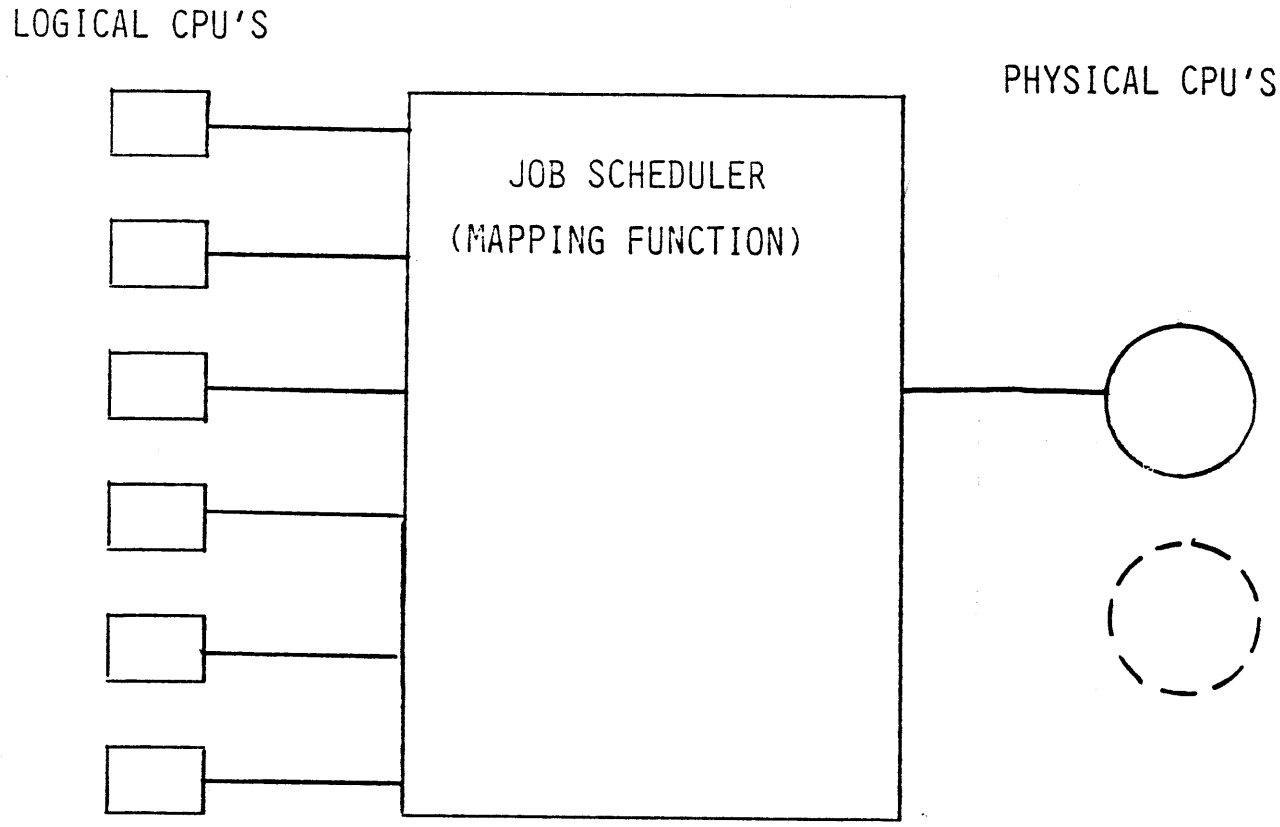
LOGICAL CPU

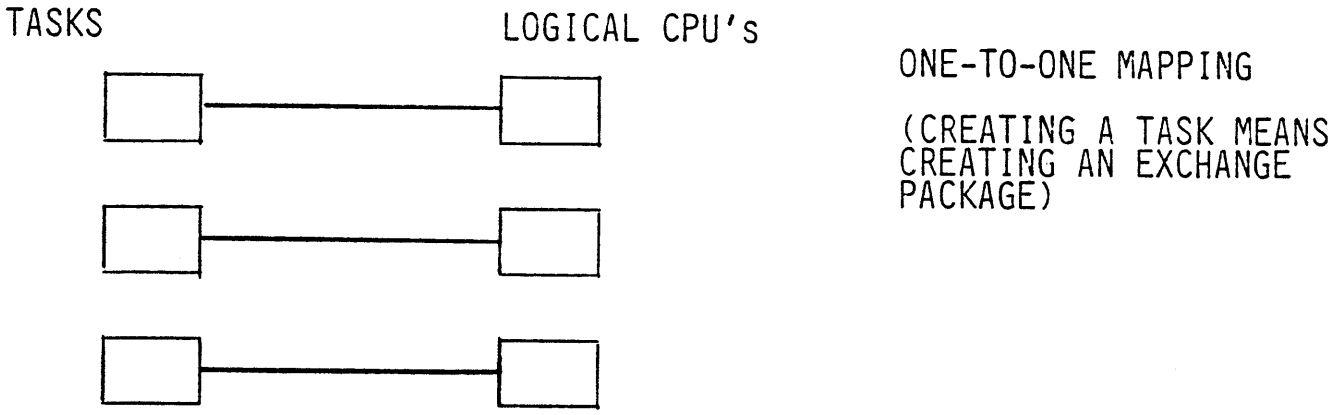
A LOGICAL CPU IS THE SCHEDULING UNIT OF AN OPERATING SYSTEM. WHEN THERE ARE MORE JOBS OR TASKS IN THE SYSTEM THAN THERE ARE REAL PROCESSORS (PHYSICAL CPU'S) THESE TASKS CAN BE ASSIGNED A LOGICAL CPU AND SCHEDULED. IT IS A FUNCTION OF THE OPERATING SYSTEM TO SCHEDULE PHYSICAL CPU'S LOGICAL CPU'S. THIS IS NOT A ONE-TO-ONE MAPPING IN A MULTIPROGRAMMING ENVIRONMENT.

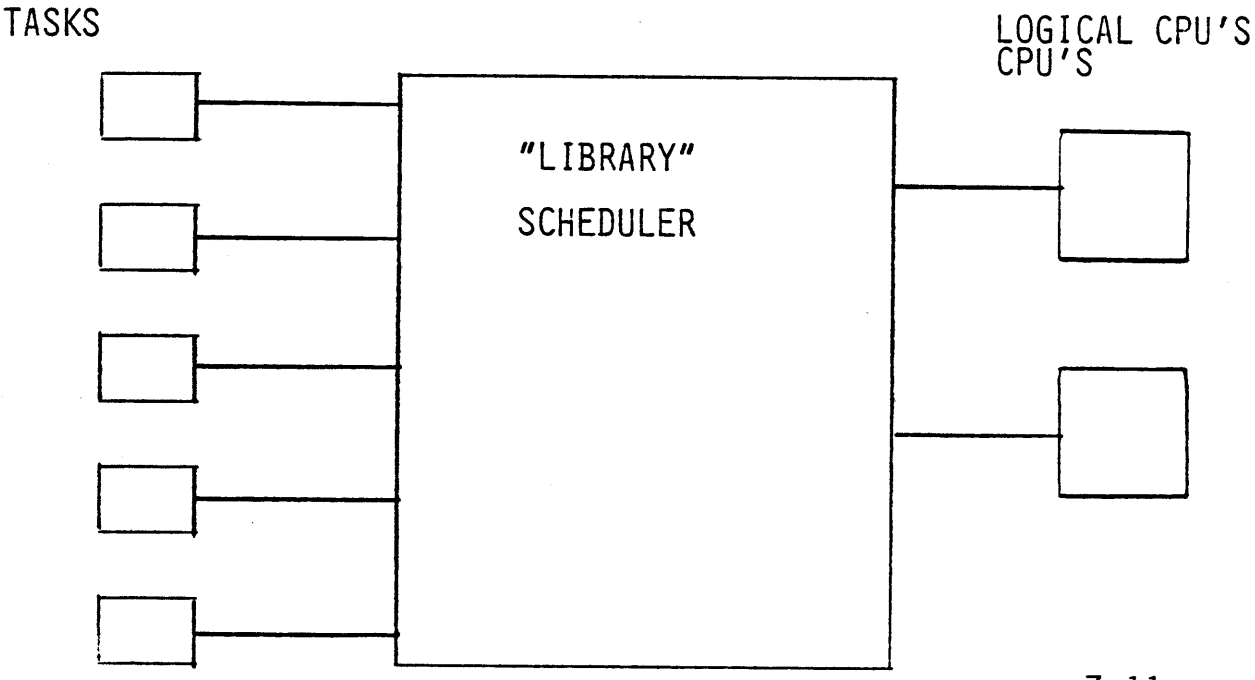
THE MAPPING OF LOGICAL CPU'S ONTO TASKS NEED NOT BE ONE-TO-ONE BUT MAY BE.

LOGICAL CPU IS SYNONOMOUS WITH:

SCHEDULING UNIT OF THE OPERATING SYSTEM
A VIRTUAL PROCESSOR
AN ENTRY IN THE JOB EXECUTION TABLE OF COS 1.11


LOGICAL CPU'S
PHYSICAL CPU'S
JOB SCHEDULER
(MAPPING FUNCTION)


TASKS
LOGICAL CPU's
ONE-TO-ONE MAPPING
(CREATING A TASK MEANS CREATING AN EXCHANGE PACKAGE)


TASKS
LOGICAL CPU'S
CPU'S
"LIBRARY"
SCHEDULER

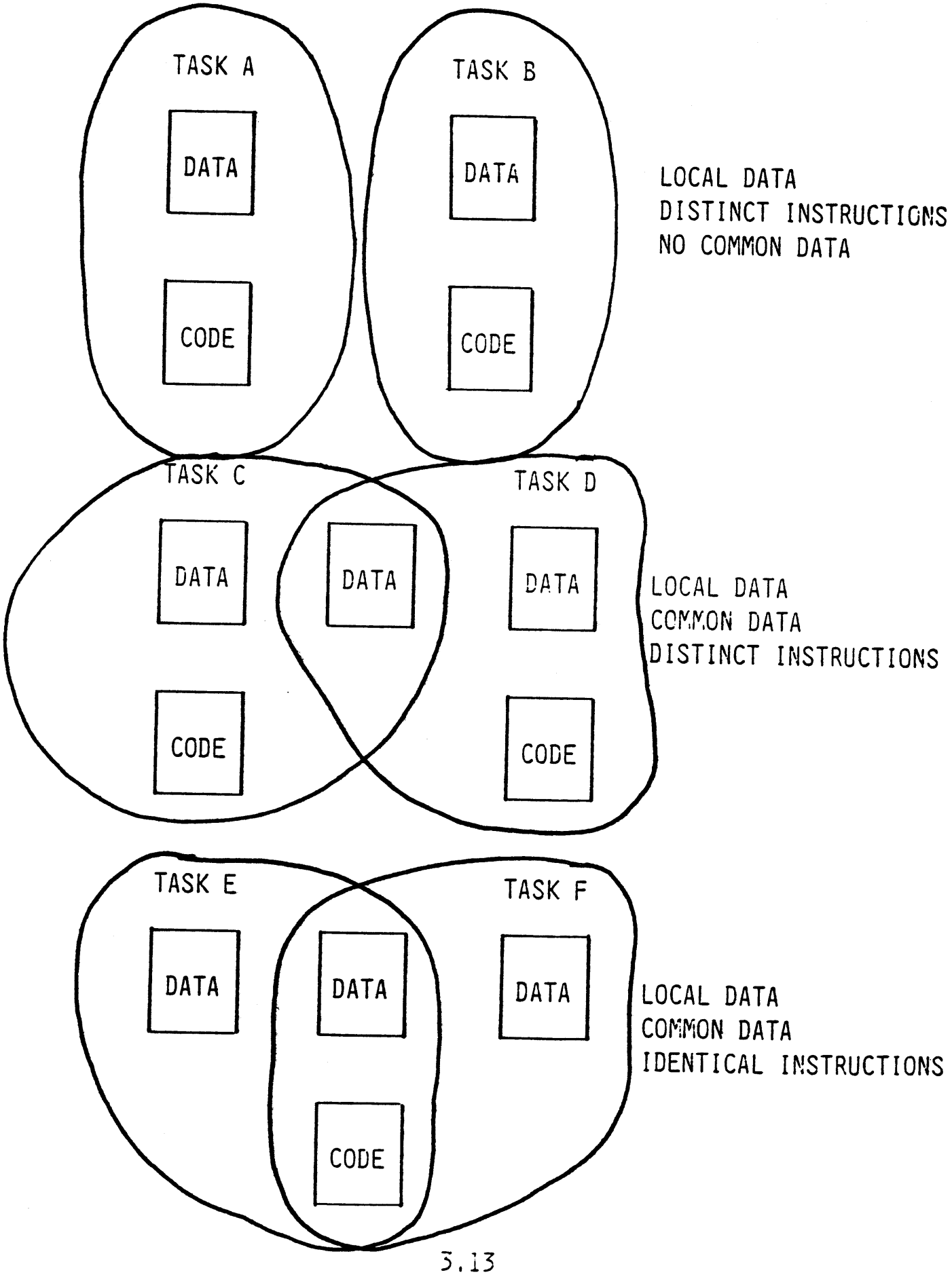
THE SCOPE OF ALL OF THE VARIABLES IN TASKS THAT ARE TO BE RUN IN PARALLEL IS IMPORTANT.

EACH TASK IS COMPRISED OF EXECUTABLE INSTRUCTIONS AND A WELL DEFINED SET OF DATA UPON WHICH THE INSTRUCTIONS ACT. THE SET OF DATA CORRESPONDING TO A TASK CAN BE DIVIDED INTO TWO SUBSETS. ONE SUBSET THAT IS DATA WHICH IS LOCAL TO THE TASK (I.E., IS DEFINED AND THEREFORE ACCESSABLE ONLY IN THAT TASK): AND ANOTHER SUBSET THAT IS COMPRISED OF DATA WHICH IS COMMON TO IT AND AT LEAST ONE OTHER TASK (I.E., DEFINED AND ACCESSABLE BY OTHER TASKS AS WELL).

ALL COMMUNICATION BETWEEN TASKS MUST BE DONE BY MEANS OF DATA THAT IS COMMON TO THOSE TASKS. DATA THAT IS TO BE WORKED ON BY MORE THAN ONE TASK (EG., A LARGE ARRAY THAT IS PROCESSED) MUST ALSO BE INCLUDED IN THE SET OF COMMON DATA.

VARIABLES USED IN THE INTERNAL FUNCTIONING OF A TASK (EG., COUNTERS AND OTHER CONTROL VARIABLES) MUST BE INCLUDED IN THE SET OF LOCALLY DEFINED DATA. THESE VARIABLES ARE DEFINED BEFORE THEY ARE USED IN THE TASKS CODE.

THE SCOPE OF VARIABLES



CRITICAL REGION

A CRITICAL REGION IS A SEGMENT OF SEQUENTIAL CODE WHICH ACCESSES SHARED PORTIONS OF MEMORY. INDETERMINATE RESULTS CAN ARISE FROM MORE THAN ONE TASK READING OR WRITING TO THE SAME MEMORY LOCATIONS SIMULTANEOUSLY. NEITHER TASK CAN BE SURE THAT THE DATA IT IS READING IS WHAT IT EXPECTED IT TO BE.

CRITICAL REGION REFERS TO CODE THAT MUST HAVE UNIQUE ACCESS TO DATA DURING ITS EXECUTION.

COMMON MEMORY IS A PART OF MEMORY KNOWN TO BOTH TASKS.

SHARED MEMORY IS MEMORY COMMON TO BOTH TASKS AND ACCESSED BY BOTH TASKS.

CRITICAL REGIONS OF CODE MUST BE "MONITORED" IF THE PROGRAM MODULES CONTAINING THEM ARE TO RUN IN PARALLEL. THIS "MONITORING" CAN BE DONE BY HAVING ONE CODE SEGMENT "LOCK" ALL OTHERS OUT OF ACCESSING SHARED MEMORY.

```
SUBROUTINE CHANGE
COMMON/BLOCK/ARRAYA,ARRAYB
    .
    .
    .

READ*,ARRAYA
    .                    THIS IS A
    .                    CRITICAL REGION
    .

WRITE*,ARRAYA
    .
    .
    .

END

SUBROUTINE REFER
COMMON/BLOCK/ARRAYA,ARRAYB          !ARRAYA & ARRAYB ARE COMMON TO BOTH
    .
    .
    .
READ*,ARRAYA                        !ONLY ARRAYA IS SHARED
    .
    .
    .
END
```

EXAMPLE 1

EXAMPLE 2

ARGUMENT LISTS VS COMMON BLOCKS

DUMMY ARGUMENTS

STATEMENT FUNCTIONS, FUNCTION SUBPROGRAMS, AND SUBROUTINE SUBPROGRAMS USE DUMMY ARGUMENTS TO INDICATE THE TYPES OF ACTUAL ARGUMENTS AND WHETHER EACH IS A SINGLE VALUE, AN ARRAY OF VALUES, OR A PROCEDURE.

EACH DUMMY ARGUMENT IS CLASSIFIED AS A VARIABLE, ARRAY, OR PROCEDURE. A DUMMY ARGUMENT NAME CAN APPEAR WHEREVER AN ACTUAL NAME OF THE SAME CLASS AND TYPE CAN APPEAR, EXCEPT WHERE EXPLICITLY PROHIBITED.

DUMMY ARGUMENT NAMES OF TYPE INTEGER CAN APPEAR AS ADJUSTABLE DIMENSION DECLARATORS IN DUMMY ARRAY DECLARATORS. A DUMMY ARGUMENT NAME CANNOT APPEAR IN AN EQUIVALENCE, DATA, SAVE, INTRINSIC, OR PARAMETER STATEMENT, AS A POINTEE IN A POINTER STATEMENT, OR IN A COMMON STATEMENT EXCEPT AS COMMON BLOCK NAMES.

COMMON BLOCKS

A COMMON BLOCK PROVIDES A MEANS OF COMMUNICATION BETWEEN EXTERNAL PROCEDURES OR BETWEEN A MAIN PROGRAM AND AN EXTERNAL PROCEDURE. THE VARIABLES AND ARRAYS IN A COMMON BLOCK CAN BE DEFINED AND REFERENCED IN ALL SUBPROGRAMS THAT CONTAIN A DECLARATION OF THAT COMMON BLOCK.

BECAUSE ASSOCIATION IS BY STORAGE SEQUENCE RATHER THAN BY NAME, THE NAMES AND TYPES OF THE VARIABLES AND ARRAYS CAN BE DIFFERENT IN THE DIFFERENT SUBPROGRAMS. A REFERENCE TO A DATUM IN A COMMON BLOCK IS PROPER IF THE DATUM IS DEFINED AND OF THE SAME TYPE AS THE TYPE OF THE NAME USED TO REFERENCE THE DATUM. HOWEVER, AN INTEGER VARIABLE THAT HAS BEEN ASSIGNED AN EXECUTABLE STATEMENT LABEL MUST NOT BE REFERENCED IN ANY PROGRAM UNIT OTHER THAN THE ONE IN WHICH IT WAS ASSIGNED.

ARGUMENT LISTS VS COMMON BLOCKS
(CONT.)

THE ONLY DIFFERENCE IN DATA TYPE PERMITTED BETWEEN THAT DEFINED AND THAT REFERENCED IS THAT EITHER PART OF A COMPLEX DATUM CAN BE REFERENCED AS A REAL DATUM.

IN A SUBPROGRAM THAT HAS DECLARED A NAMED OR BLANK COMMON BLOCK, THE ENTITIES IN THE BLOCK REMAIN DEFINED AFTER THE EXECUTION OF A RETURN OR END STATEMENT.

RESTRICTIONS ON THE ASSOCIATION OF ENTITIES

IF A SUBPROGRAM REFERENCE CAUSES A DUMMY ARGUMENT TO BECOME ASSOCIATED WITH AN ENTITY IN A COMMON BLOCK IN THE REFERENCED SUBPROGRAM, NEITHER THE DUMMY ARGUMENT NOR THE ENTITY IN THE COMMON BLOCK CAN BECOME DEFINED WITHIN THE SUBPROGRAM. FOR EXAMPLE, IF A SUBROUTINE CONTAINING STATEMENTS

```
      SUBROUTINE XYZ (A)

       COMMON C
```

IS REFERENCED BY A PROGRAM UNIT THAT CONTAINS THE STATEMENTS

```
      COMMON B

      CALL XYZ (B)
```

THE DUMMY ARGUMENT A BECOMES ASSOCIATED WITH THE ACTUAL ARGUMENT B. B AND C ARE ASSOCIATED IN A COMMON BLOCK. NEITHER A NOR C CAN BECOME DEFINED DURING THE EXECUTION OF SUBROUTINE XYZ OR BY ANY PROCEDURES IT REFERENCES.

REENTRANT VS SERIALLY REUSABLE

SERIALLY REUSABLE - THE PROPERTY OF AN INSTRUCTION STREAM (CODE SEGMENT) THAT ALLOWS ONE COPY OF IT TO BE USED BY MORE THAN ONE JOB OR TASK BUT ONLY ONE AT A TIME. THE SECOND TASK WISHING TO ENTER A SERIALLY REENTRANT CODE MUST WAIT IF ANOTHER USER HAS ENTERED FIRST AND NOT YET EXITED. THE ROUTINES ENVIRONMENT MUST BE RESTORED TO ITS INITIAL CONDITION AFTER EACH USE. THIS IS REFERRED TO AS SINGLE THREADING OF THE CODE. I/O ROUTINES WILL BE SERIALLY REUSABLE IN COS 1.13.

SINGLE THREADING - SUPPORTING ONLY ONE USER AT A TIME.

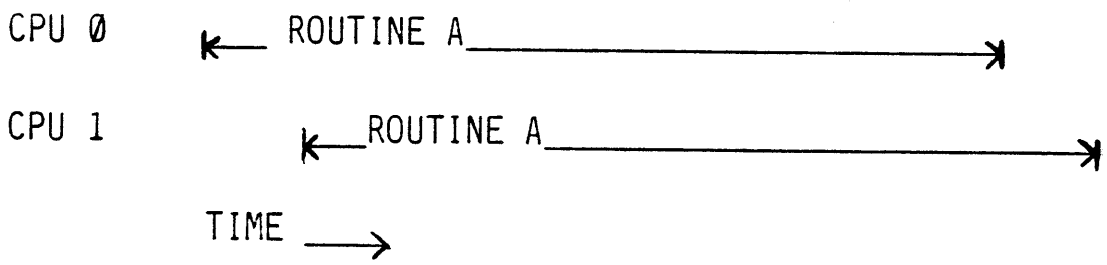
REENTRANT - THE PROPERTY OF A PROGRAM MODULE THAT ALLOWS ONE COPY OF IT TO BE USED BY MORE THAN ONE JOB OR TASK. A MECHANISM IS SUPPLIED BY WHICH THE ROUTINES ENVIRONMENT IS PRESERVED, I.E., LOCAL VARIABLES AND CONTROL INDICATORS ARE ASSIGNED INDEPENDENT STORAGE LOCATION EACH TIME THE ROUTINE IS CALLED. IN COS 1.13 THIS IS DONE USING A STACK STRUCTURE.

LOCAL VARIABLE - A VARIABLE WHOSE VALUE IS KNOWN ONLY TO THE PROGRAM MODULE IN WHICH IT IS DEFINED.

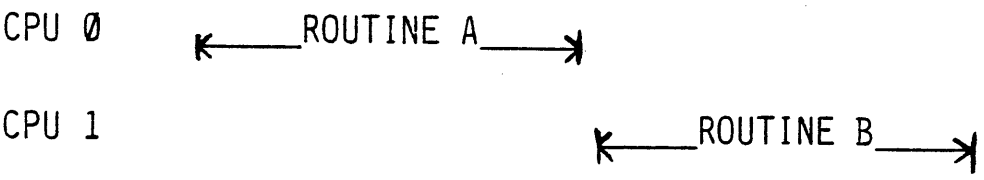
STACK - A DATA STRUCTURE PROVIDING A DYNAMIC SEQUENTIAL DATA LIST HAVING SPECIAL PROVISIONS FOR ACCESS FROM ONE END OR THE OTHER. A LAST IN, FIRST OUT (PUSH DOWN, POP UP) STACK IS ACCESSED FROM JUST ONE END.

A PROGRAM MODULE MUST BE REENTRANT IF IT IS TO BE DESIGNATED AS THE CODE OF TASKS THAT CAN BE RUN IN PARALLEL. CRITICAL REGIONS OF THE CODE MUST STILL BE MONITORED.

ROUTINE A IS REENTRANT



ROUTINE B IS SERIALLY REUSABLE



SECTION 4

CRAY'S MULTITASKING FACILITIES

OBJECTIVE: AT THE END OF THE COURSE THE LEARNER IS ABLE TO WRITE A FORTRAN OR CAL PROGRAM MAKING USE OF THE MULTITASKING FACILITIES SUPPLIED BY CFT LIBRARIES AND COS 1.13.

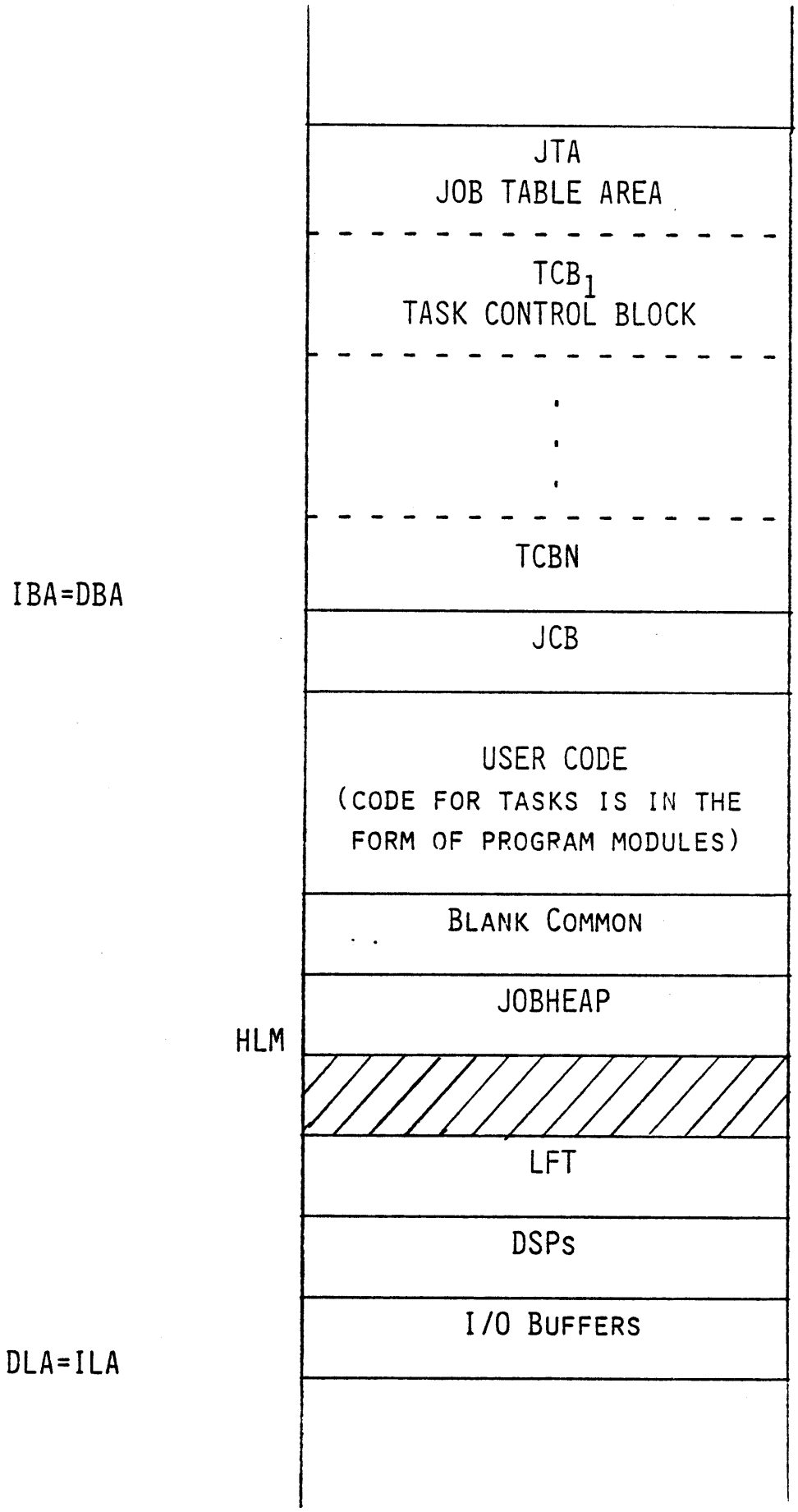
THE OPERATING SYSTEM ENVIRONMENT

THE PROGRAMMER'S VIEW OF A JOB WILL REMAIN MUCH THE SAME AS BEFORE: THE CONTROL CARDS ARE EXECUTED SEQUENTAILLY, ONE AT A TIME; IF A CONTROL CARD INDICATES THAT USER CODE IS TO BE INVOKED, IT IS LOADED INTO MEMORY AND AN EXCHANGE JUMP PASSES CONTROL TO THE USER'S CODE. WHEN THE USER'S PROGRAM COMPLETES AN EXCHANGE JUMP , THE OPERATING SYSTEM WILL CAUSE THE NEXT, SEQUENTIAL, CONTROL CARD TO BE EXECUTED.

IT IS DURING THE EXECUTION OF A USER'S CODE THAT INSTRUCTIONS CAN BE ISSUED WHICH WILL CAUSE ADDITIONAL CPU RESOURCES TO BE MADE AVAILABLE FOR THE EXECUTION OF THAT CODE. THIS IS ACCOMPLISHED BY PARTITIONING THE USER'S CODE INTO A SET OF TASKS. IT IS THE PROGRAMMER'S RESPONSIBILITY TO CREATE THESE PARTITIONS, CONTROL THEIR INTERACTION, AND CAUSE THEM TO COMPLETE AT THE APPROPRIATE TIME. SINCE THE OPERATING SYSTEM VIEWS THE EXECUTING CODE AS A SINGLE, SEQUENTIAL, CONTROL CARD STEP, IT WILL NOT ALLOW EXECUTION OF THE NEXT CONTROL CARD UNTIL ALL ACTIVITY IN THE USER'S CODE TERMINATES. THIS WILL ONLY HAPPEN WHEN ALL THE PROGRAMMER'S TASKS HAVE COMPLETED OR WHEN A TASK ENCOUNTERS A FATAL ERROR AND THE OPERATING SYSTEM FORCES ALL ACTIVITY TO TERMINATE.

TASK CONTROL BLOCK - THE AREA IN USER ASSIGNED MEMORY (BUT NOT ACCESSABLE TO THE USER JOB) CONTAINING ALL THE INFORMATION ASSOCIATED WITH AN ACTIVE TASK (ONE THAT HAS BEEN STARTED BUT HAS NOT YET ENCOUNTERED THE STOP OR RETURN). THE CONTENTS INCLUDE: THE TASKS EXCHANGE PACKAGE, POINTERS TO THE TASKSTACK AND SUBROUTINES CONTAINING TASK CODE.

USER AREA IN MEMORY (COS 1.13)



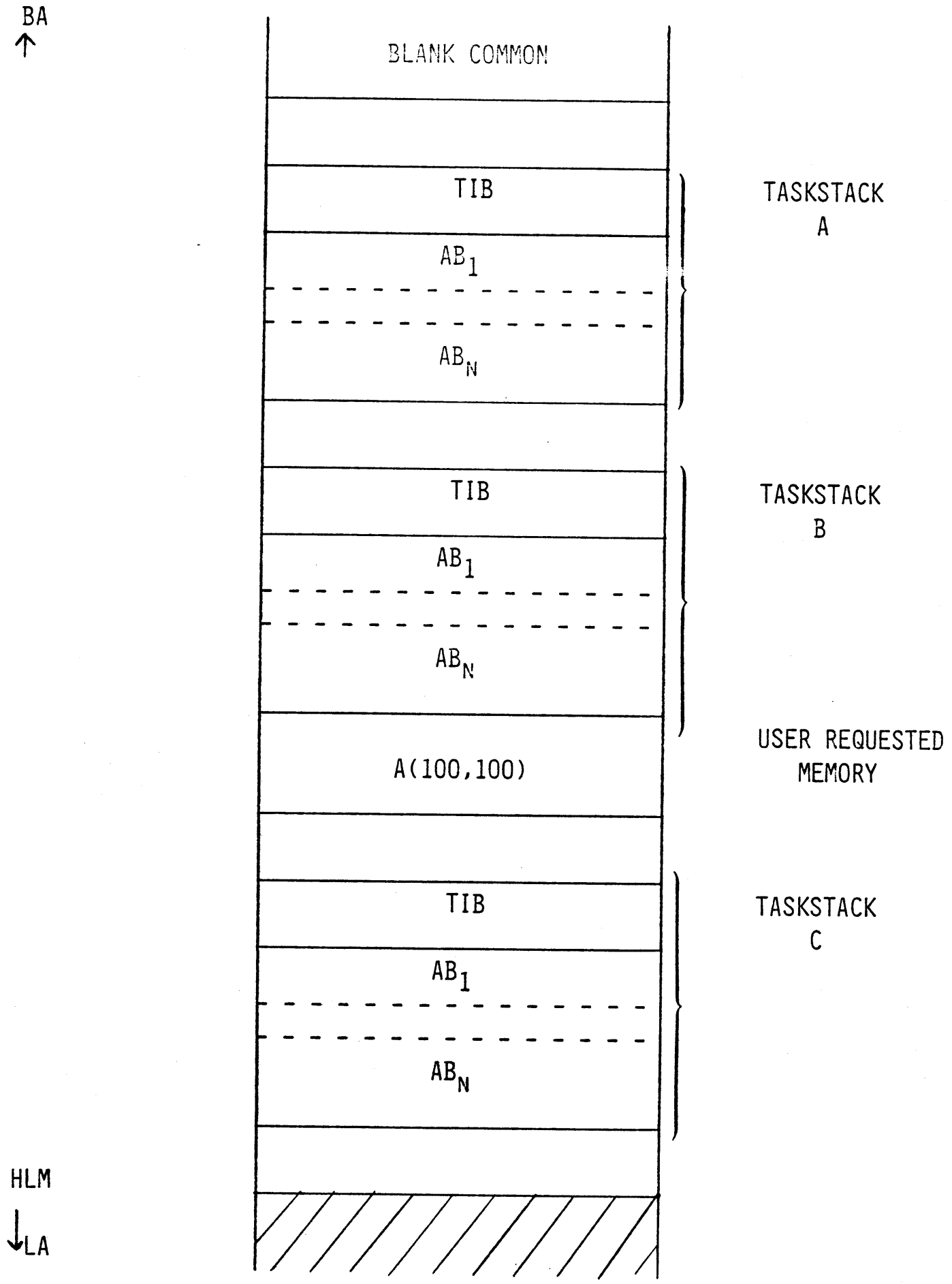
JOBHEAP - THE AREA IN USER MEMORY BETWEEN BLACK COMMON AND HLM THAT CAN BE DYNAMICALLY ALLOCATED BY LIBRARY ROUTINES TO TASKSTACKS AND USER MEMORY REQUESTS.

TASKSTACK - A PUSH DOWN, POP UP STACK CREATED UPON THE ACTIVATION (FIRING UP) OF A TASK. THE ELEMENTS OF THE TASKSTACK ARE ACTIVATION BLOCKS. ONE ACTIVATION BLOCK IS CREATED (PLACED ON TOP) EACH TIME A SUBROUTINE IS CALLED AND POPPED OFF WHEN STOP OR RETURN IS EXECUTED.

ACTIVATION BLOCK (AB) - THE ELEMENT OF A TASKSTACK ASSOCIATED WITH A SUBROUTINE CALL FROM WITHIN THE TASK. AN ACTIVATION BLOCK CONTAINS: B AND T REGISTERS SAVE SPACE, LOCAL VARIABLE STORAGE LOCATIONS, AND SPACE FOR ARGUMENT LISTS.

TASK INFORMATION BLOCK (TIB) - A BLOCK OF DATA LOCATED AT THE BASE OF THE TASKSTACK. IT CONTAINS INFORMATION INCLUDING: TASK ID, TASKVALUE, CPU#, (POINTER TO LIBRARY-SCHEDULER QUEUES). THE TIB IS NOT AN ELEMENT OF THE TASKSTACK.

JOBHEAP


BA
BLANK COMMON
TIB
AB1
ABN
TASKSTACK A
TIB
AB1
ABN
TASKSTACK B
A(100,100)
USER REQUESTED MEMORY
TIB
AB1
ABN
TASKSTACK C
HLM
LA

THE FORTRAN ENVIRONMENT

THE PROGRAMMER WHO USES MULTI-TASKING MUST BE AWARE OF SEVERAL DIFFERENCES BETWEEN THE PROGRAM ENVIRONMENT THESE FACILITIES REQUIRE AND THE PREVIOUS CFT ENVIRONMENT. THE CRAY IMPLEMENTATION OF MULTITASKING HAS REQUIRED THAT:

1. THAT CFT USE THE NEW CALLING SEQUENCE.
2. THAT CFT GENERATE RE-ENTRANT CODE.
3. THAT IT BE POSSIBLE TO ALLOCATE SEPARATE LOCAL STORAGE FOR EACH TASK.
4. THAT THE SIZE OF BLANK COMMON REMAIN THE SAME THROUGHOUT EXECUTION OF THE PROGRAM. A NEW MEMORY MANAGEMENT ROUTINE WILL BE MADE AVAILABLE SO THAT A PROGRAM MAY REQUEST AND FREE STORAGE AS IS NEEDED.

THIS FACILITY PROVIDES ONLY THE BASIC FUNCTIONS NEEDED TO CONSTRUCT PROGRAMS. THE PROGRAMMER MUST DETERMINE THE BEST APPROACH FOR UTILIZING THE FACILITIES ON THE BASIS OF THE DESIRED PROGRAM STRUCTURE. THE FOLLOWING IMPORTANT ITEMS WILL NEED TO BE CONSIDERED:

1. THIS FACILITY DOES NOT INDICATE A RELATIONSHIP BETWEEN TASKS. THE DECISION TO USE CO-ROUTINES OR PARENT-CHILD RELATIONSHIPS MUST BE MADE BY THE PROGRAMMER.
2. THIS FACILITY DOES NOT PROVIDE AN EXPLICIT COMMUNICATION PATH BETWEEN TASKS. THE PROGRAMMER CAN USE EVENTS OR PASS INFORMATION THROUGH PARAMETERS OF COMMON IF INTERACTION AT THIS LEVEL IS DESIRED.
3. THIS FACILITY OFFERS NO SUPPORT FOR OVERLAYS. GREAT CARE MUST BE TAKEN IF MULTI-TASKING AND OVERLAYS ARE TO BE USED IN THE SAME PROGRAM. THE PROGRAMMER MUST STRUCTURE THE PROGRAM SO THAT ALL POTENTIAL CONFLICTS ARE AVOIDED.
4. THIS FACILITY PROVIDES NO DIRECT PROTECTION FOR CRITICAL DATA. THE PROGRAMMER MUST MONITOR ACCESS OF SHARED MEMORY BY USING LOCKS TO PROTECT CRITICAL REGION OF CODE.

MULTITASKING PROCEDURES

CATEGORY	NAME	FUNC TYPE	DESCRIPTION	ARGUMENT TYPES X_1	X_2	X_3	
TASK CONTROL	TSKSTART		Initiates a task	Task Control Array	Sub-Routine Name	Arg. List	
	TSKWAIT		Wait for the completion of a task	Task Control Array			
	TSKTEST	L	Determines if the task exists	Task Control Array			
	TSKVALUE		Returns the programmer assigned task value	I			
LOCK CONTROL	LOCKASGN		Creates a unique lock identifier	I			
	LOCKON		Sets a lock	I			
	LOCKOFF		Clears a lock	I			
	LOCKTEST	L	Determines if a lock has already been set	I			
	LOCKREL		Releases the lock identifier	I			
EVENT CONTROL	EVASGN		Allocates a unique identifier and causes an event to become defined	I			
	EVWAIT		Waits for an event to be posted by another task	I			
	EVPOST		Posts an event	I			
	EVCLEAR		Clears an event removing it from-a posted state	I			
	EVTEST	L	Determines if an event has been posted	I			
	EVREL		Releases the event identifier	I			

TASKS

TO INITIATE A TASK THE PROGRAMMER NEEDS THE FOLLOWING STATEMENT:

```
CALL TSKSTART ( TASK-CONTROL-ARRAY ,
                SUBROUTINE-NAME ,
                ARGUMENT-LIST )
```

THE TASK-CONTROL-ARRAY MUST BE UNIQUE FOR EACH ACTIVE TASK THE USER CREATES. THIS ARRAY CAN CONTAIN CONTROL INFORMATION THAT IS USED BY THE MULTITASKING LIBRARY TO CONTROL THE EXECUTION OF THE SPECIFIC TASK. TWO ENTRIES ARE CURRENTLY RESERVED IN THIS ARRAY: THE FIRST WORD OF THE ARRAY MUST CONTAIN THE TOTAL NUMBER OF WORDS IN THE ARRAY (AT LEAST 2); THE SECOND WORD IS FILLED IN BY THE MULTITASKING LIBRARY WITH THE UNIQUE TASK IDENTIFIER. THE OPTIONAL THIRD WORD IS A USER SUPPLIED VALUE THAT WILL BE ASSOCIATED WITH THE NEW TASK AND CAN BE RETRIEVED BY THAT ROUTINE USING THE TSKVALUE CALL. THE FIRST WORD OF THIS ARRAY SHOULD BE SET TO 3 IF A TASK VALUE IS TO BE PASSED. THE PROGRAMMER WILL NEED TO USE THIS CONTROL ARRAY WHEN WAITING ON TASK COMPLETION OR DETERMINING IF A TASK IS STILL EXECUTING.

THE SUBROUTINE-NAME IS THE EXTERNAL ENTRY POINT THAT CONTAINS THE CODE FOR THE TASK. BECAUSE OF THE DESIGN OF THE FORTRAN LANGUAGE, THE PROGRAMMER WILL ALSO NEED AN EXTERNAL STATEMENT IN THE PROGRAM FOR THIS ENTRY POINT.

THE ARGUMENT-LIST IS THE LIST OF PARAMETERS THAT NEEDS TO BE PASSED TO THE NEW TASK WHEN THE SUBROUTINE-NAME IS ENTERED. WHAT, IF ANYTHING, IS PASSED MUST BE DETERMINED BY THE PROGRAMMER.

TSKSTART MAKES 1 TASK READY.

```
      PROGRAM ONETASK    (Main Program is one task, subroutine is another)
      EXTERNAL B
      LOGICAL DO
      INTEGER READY,COMPLETE,BTCA
      DIMENSION ARRAY(128,128),BTCA(3)
      COMMON/MSG1/DO,ARRAY,READY,COMPLETE
      CALL EVASGN(READY)
      CALL EVASGN(COMPLETE)
      DO=.TRUE.
      BTCA(1)=3
            .
      CALL TSKSTART(BTCA,B...)
 10   CALL EVPOST(READY)              ! signal go for task
      IF(DO.NE TRUE)GO TO 20
C           Process A: red portion of ARRAY
            .
            .
            .
      CALL EVWAIT(COMPLETE)           ! wait for task B to complete one task
      CALL EVCLEAR(COMPLETE)
C           Test Process: convergence
            .
            .
            .
            .
C        if the test shows no more processing necessary DO=False
      GO TO 10                        ! task must turn itself off when it is done
 20   CONTINUE
            .
            .
            .
            .
      END


C       Subroutine to be "fired up" as a task
      SUBROUTINE B( ,...)
      COMMON/MSG1/DO,A(128,128),READY,COMPLETE
      LOGICAL DO
 5    CALL EVWAIT(READY               ! wait to do next pass
      CALL EVCLEAR (READY)            ! once through at a time
      IF(DO.NE.TRUE)STOP              ! task is deactivated by stop
C           Process B: black portion of matrix A
            .
            .
            .
      CALL EVPOST(COMPLETE)           ! signal "ready for test"
      GO TO 5
      END
```

2.1.2 TSKWAIT - WAIT FOR TASK COMPLETION

THE TSKWAIT ROUTINE WAITS FOR A SPECIFIC TASK TO COMPLETE EXECUTION. THIS FUNCTION IS INVOKED BY THE FOLLOWING CFT STATEMENT:

```
CALL TSKWAIT(TASK-CONTROL-ARRAY)
```

THE TASK CONTROL ARRAY MUST CONTAIN THE SAME INFORMATION IT HAD ON RETURN FROM THE TSKSTART CALL THAT CREATED THE TASK NOW BEING AWAITED. THE TASK EXECUTING THIS CALL WILL BE SUSPENDED UNTIL THE TASK BEING AWAITED HAS COMPLETED EXECUTION. A TASK COMPLETES BY EXECUTING A STOP, END, RETURN OR CALL EXIT. MORE THAN ONE TASK CAN BE WAITING ON THE COMPLETION OF A GIVEN TASK. ALL WAITING TASKS WILL THEN BE READIED BY COMPLETION.

```
FILE: TESTMLT2 JCL      A   CRAY RESEARCH INC. COMPUTER SERVICES.DEVELOPMEN

JOB.JN=JMNMULT.M.
ACCOUNT.AC=3001569.
ACCESS.DN=CFT.ID=NEWSEQ.
ACCESS.DN=LDR.ID=STACK.
ACCESS.DN=CAL.ID=NEWSEQ.
ACCESS.DN=$SYSLIB.ID=STACK.
ACCESS.DN=$FTLIB.ID=STACK.
ACCESS.DN=$SCILIB.ID=STACK.
ACCESS.DN=MULTLIB.ID=JMN.
ACCESS.DN=STKLIB.ID=JMN.
CFT.
LDR.LIB=MULTLIB:STKLIB.STK=4000:2000.MM=100000:5000.MAP.
EXIT.
DUMPJOB.
DUMP.LW=15000.
/EOF
      PROGRAM TESTMULT
C
      PARAMETER (SIZE=10)
      PARAMETER (NTASKS=10)
C
      REAL A(SIZE,SIZE),B(SIZE,SIZE),C(SIZE,SIZE)
      INTEGER ITCA(2,NTASKS)
C
      EXTERNAL MXM
      DATA (ITCA(1,I),I=1,NTASKS)/NTASKS*2/
```

```
C
      DO 20 I=1,SIZE
      DO 10 J=1,SIZE
      B(I,J)=I+J
      A(I,J)=0.0
10    CONTINUE
      A(I,I) = 1.0
20    CONTINUE
C
      IF ( SIZE .LE. 10 ) THEN
        PRINT 100,((A(I,J),I=1,SIZE),J=1,SIZE)
        PRINT 110
        PRINT 100,((B(I,J),I=1,SIZE),J=1,SIZE)
        PRINT 110
100     FORMAT(//,10(1X,10F12.2,/))
110     FORMAT(' ')
      ENDIF
C
      L=SIZE
      M=SIZE
      N=SIZE/NTASKS
C
      DO 30 I=1,NTASKS-1
      J = I*N+1
      ITSTRT = IRTC()
      CALL TSKSTART(ITCA(1,I),MXM,A,L,B(1,J),M,C(1,J),N)
C     CALL MXM(A,L,B(1,J),M,C(1,J),N)
      ITEND = IRTC() - ITSTRT
      ITTOT = ITTOT + ITEND

      PRINT 1000,ITEND,I
1000  FORMAT(' IT TOOK ',I8,' CLOCKS TO START TASK ',I4)
30    CONTINUE
C
      PRINT 1100,ITTOT/(NTASKS-1)
1100  FORMAT(' IT TOOK ',I8,' CLOCKS ON THE AVERAGE TO START A TASK')
C
      ITTOT = 0
C
      CALL MXM(A,L,B,M,C,N)
C
      DO 40 I=1,NTASKS-1
      ITSTRT = IRTC()
      CALL TSKWAIT(ITCA(1,I))
      ITEND = IRTC() - ITSTRT
      ITTOT = ITTOT + ITEND
      PRINT 1200,ITEND,I
1200  FORMAT(' IT TOOK ',I8,' CLOCKS TO WAIT FOR TASK ',I4)
40    CONTINUE
C
      PRINT 1300,ITTOT/(NTASKS-1)
1300  FORMAT(' IT TOOK ',I8,' CLOCKS ON THE AVERAGE TO AWAIT A TASK')
C
      IF ( SIZE .LE. 10 ) THEN
        PRINT 100,((C(I,J),I=1,SIZE),J=1,SIZE)
        PRINT 110
      ENDIF
C
      STOP
      END
/EOF
```

To determine if a task exists the programmer can use the function:

TSKTEST (task-control array)

The task-control-array must contain the same information it had on return from the initial TSKSTART call.

This function will return a logical value, so the programmer must also include a LOGICAL type declaration for it in the program.

A logical "TRUE" is returned if a task exists with an identifier that matches the task identifier in the task-control-array. This result will be returned no matter what state of execution the task is in.

A logical "FALSE" is returned if the task was never created or has completed execution.

A task can retrieve its value by executing the following statement:

CALL TSKVALUE (return-value)

The multitasking routine will return the value to the task by placing it in the dummy argument "return-value".

The TSKVALUE subroutine will return a ZERO (0) value if no value has been provided by the Task Control Array. This happens if the Task Control Array was less than three words in length or if the task is the initial root task created by the loader.

The third word of the Task Control Array will be reserved for use by the programmer. If programs are to use this word, the Task Control Array must be extended to three words in length. If programs don't use the word, the length may remain at two words (the length of the array is stored in the first word of the array).

The user may place any value desired into this third word. However, the value must be placed there before the Task Control Array is used to initiate a task with a call to TSKSTART, and can not be changed during execution of the task. Suggested values include a programmer generated task identifier or name, or a pointer to a task local storage area.

This value will be saved by the TSKSTART routine in a task local data area that is used to manage the task. This implementation prevents the user from altering the value during the lifetime of the task.

A subroutine call has been chosen to avoid the automatic typing problems associated with functions. The programmer can choose the data type of the dummy argument to match that of the value initially placed in the Task Control Array.

To cause a unique lock identifier to be created the programmer needs the following statement:

```
CALL LOCKASGN ( integer-variable )
```

The multi-tasking support library will create an unused identifier that will be returned in the integer-variable, and can be used by the program as a lock identifier to set, clear, or test a lock.

To set a lock the programmer needs the following statement:

```
CALL LOCKON ( integer-variable )
```

The integer-variable will be set with a unique value that indicates that the lock is in the locked state. This variable must have been initialized by a previous call to LOCKASGN.

The function of this routine is to set a lock and return execution to the calling program. If the lock has already been set, the calling program is suspended until the lock has been cleared by another task and can be set by this one.

To clear a lock the programmer needs the following statement:

```
CALL LOCKOFF ( integer-variable )
```

If there are other tasks waiting on this lock, only one of them (the one that has been waiting the longest) is readied, allowing it to set the lock and proceed with processing in its critical region.

The integer-variable will be set with a unique value that indicates that the lock is in the unlocked state. This variable must have been initialized by a previous call to LOCKASGN.

Shared portions of memory are monitored by protecting a critical region of code by means of a lock.

```
CALL LOCKON (INT)

 critical region

CALL LOCKOFF (INT)
```

```
      SUBROUTINE qinitial
c
c initialize the queuing mechanism.
c
c the queue and the controlling data structures are in a common block.
c the following data items are needed:
c   qlock : a lock used to limit access to the queue and associated
c           control structures. This is used in place of calls to
c           EVTEST to determine the presence of messages in the queue
c           and the availability of room for new messages
c   qnummsg : a counter  indicating the number of messages in the queue
c           that have not been received.
c   qhasmsg : an event that signals the presence of 1 or more messages
c           in the queue.
c   qhasroom : an event that can be used when the queue is full to wait
c           for room to become available for new messages.
c   qinput : an index into the queue that addresses the last message
c           that was added to the queue.
c   qoutput : an index into the queue that addresses the last message
c           that was removed from the queue.
c   qmsg : an array, used as a circular buffer, that represents the queue
c   qlength : the number of entries that may be made in the queue.
c
      INTEGER qlength
      PARAMETER(qlength = 100)
      COMMON /aq/qlock,qnummsg,qhasmsg,qhasroom,qinput,qoutput,
     1           qmsg(qlength)
      INTEGER qlock,qnummsg,qhasmsg,qhasroom,qinput,qoutput,qmsg

      CALL LOCKASGN(qlock)
      CALL EVASGN(qhasroom)
      CALL EVPOST(qhasroom)
      CALL EVASGN(qhasmsg)
      CALL EVCLEAR(qhasmsg)
      qinput = 0
      qoutput = 0
      qnummsg = 0
      END
```

```
 82          c
 83          c retrieve the next item from the queue.
 84          c
 85          c the following processing steps are necessary:
 86          c   1. get exclusive access to the queue
 87          c   2. if there is nothing in the queue, wait until there is
 88          c   3. when information is available in the queue, compute its location
 89          c      and retrieve it from the circular buffer
 90          c   4. if the queue was previously full, post the event that signals
 91          c      that the queue now has an empty slot
 92          c   5. release the lock so that other tasks may have access to the queue.
 93          c
 94    2.          INTEGER qlength
 95    3.          PARAMETER(qlength = 100)
 96    4.          COMMON /aq/qlock,qnummsg,qhasmsg,qhasroom,qinput,qoutput,
 97                1           qmsg(qlength)
 98    5.          INTEGER qlock,qnummsg,qhasmsg,qhasroom,qinput,qoutput,qmsg
 99
100    6.        1 CALL LOCKON( qlock)
101    7.          IF( qnummsg .EQ. 0 ) THEN
102    8.                CALL EVCLEAR(qhasmsg)
103    9.                CALL LOCKOFF (qlock)
104   10.                CALL EVWAIT(qhasmsg)
105   11.                GOTO 1
106   12.           END IF
107
108   13.           IF( qoutput .LT. qlength ) THEN
109   14.                qoutput = qoutput + 1
110   15.           ELSE
111   16.                qoutput = 1
112   17.           END IF
113
114   18.           qnummsg = qnummsg -1
115   19.           msgfromq = qmsg(qoutput)
116   20.           IF( qnummsg .EQ. (qlength - 1) ) CALL EVPOST(qhasroom)
117
118   21.           CALL LOCKOFF(qlock)
              E     END           ON=CELMPQRSTV                    11/12/82-13:31:40      CFT X.11(11/08/82)  PA
```

```
39
40      1.        SUBROUTINE msgtoq( msg )
41              c
42              c add a message to the queue.
43              c
44              c this is accomplished by the following sequence of activities:
45              c    1. get exclusive access to the queue
46              c    2. if there is no room, release the lock and wait on an event
47              c    3. when room is available, compute the next location in the
48              c       queue that should receive the new message and store it away
49              c    4. if this is the only message in the queue, post an event so
50              c       that other tasks that are waiting for a message will
51              c       take a look in the queue
52              c    5. release exclusive access to the queue.
53              c
54      2.        INTEGER qlength
55      3.        PARAMETER(qlength = 100)
56      4.        COMMON /aq/qlock,qnummsg,qhasmsg,qhasroom,qinput,qoutput,
57               1          qmsg(qlength)
58      5.        INTEGER qlock,qnummsg,qhasmsg,qhasroom,qinput,qoutput,qmsg
59
60      6.      1 CALL LOCKON(qlock)
61      7.        IF( qnummsg .EQ. qlength )  THEN
62      8.              CALL EVCLEAR(qhasroom)
63      9.              CALL LOCKOFF(qlock)
64     10.              CALL EVWAIT(qhasroom)
65     11.              GOTO 1
66     12.        END IF
67
68     13.        IF( qinput .LT. qlength )   THEN
69     14.              qinput = qinput + 1
70     15.          ELSE
71     16.              qinput = 1
72     17.        END IF
73
74     18.        qnummsg = qnummsg + 1
75     19.        qmsg(qinput) = msg
76     20.        IF( qnummsg .EQ. 1 ) CALL EVPOST(qhasmsg)
77
78     21.        CALL LOCKOFF(qlock)
           .        END
```

To determine if a lock has already been set the programmer needs the following function:

```
LOCKTEST ( integer-variable )
```

The value in the integer-variable is tested to determine if it is in the locked state. If the variable is in the unlocked state, it is changed to the locked state. This variable must have been initialized by a previous call to LOCKASGN.

If the integer-variable was originally in the locked state, this function will return a logical "TRUE" value. If not, it is placed in the locked state and a logical "FALSE" value is returned.

Since the data type for this function does not match the normal FORTRAN default, the programmer will need a LOGICAL LOCKTEST type declaration in the program.

To release a unique identifier that has been created by a LOCKASGN call the programmer needs the following statement:

```
CALL LOCKREL ( integer-variable )
```

The value of the integer-variable must be the same value returned from the LOCKASGN call.

The main function of this routine is to detect errors that may arise when a task is waiting on a lock that will never again be cleared. Any reference to this identifier while it remains unassigned is an error, although it may again be used after another call to LOCKASGN.

```
      SUBROUTINE SAMPLE

      COMMON/SAM/'''

      LOGICAL LOCKTEST
      N=0

100   N=N+64
        .                          THIS WILL BE REPEATED WHILE
        .                          WAITING FOR THE LOCK TO BE
        .                          CLEARED.

      CALL LOCKTEST (AMPLE)
      IF (AMPLE) GO TO 100
        .
        .                          CRITICAL REGION
        .
      CALL LOCKOFF (AMPLE)
        .
        .
        .
      CALL LOCKREL (AMPLE)         !ALL FUTURE REFERENCES TO
      RETURN                       !THE LOCK AMPLE WILL BE ERRORS
```

To allocate a unique identifier and cause an event to become defined, the programmer needs the following statement:

```
CALL EVASGN ( integer-variable )
```

The multi-tasking support library will create an unused identifier that will be returned in the integer-variable and can be used by the program as an event identifier to post, clear, wait on, or test an event.

To wait for an event to be posted by another task the programmer needs the following statement:

```
CALL EVWAIT ( integer-variable )
```

The contents of the integer-variable must be the value generated by the multi-tasking support library on a call to EVASGN.

The function of this routine is to suspend execution of the task until the named event has been posted.

To post an event the programmer needs the following statement:

```
CALL EVPOST ( integer-variable )
```

The contents of the integer-variable must be the value generated by the multi-tasking support library on a call to EVASGN.

The function of this routine is to mark an event as posted and to cause all tasks waiting on that event to resume execution (i.e., this can cause multiple tasks to become ready).

To clear an event the user needs the following statement:

```
CALL EVCLEAR ( integer-variable )
```

The contents of the integer-variable must be the value generated by the multi-tasking support library on a call to EVASGN.

The function of this routine is to remove an event from the posted state.

```
      PROGRAM ONETASK    (Main Program is one task, subroutine is another)
      EXTERNAL B
      LOGICAL DO
      INTEGER READY,COMPLETE,BTCA
      DIMENSION ARRAY(128,128),BTCA(3)
      COMMON/MSG1/DO,ARRAY,READY,COMPLETE
      CALL EVASGN(READY)
      CALL EVASGN(COMPLETE)
      DO=.TRUE.
      BTCA(1)=3
            .
      CALL TSKSTART(BTCA,B...)
 10   CALL EVPOST(READY)              ! signal go for task
      IF(DO.NE TRUE)GO TO 20
C           Process A: red portion of ARRAY
            .
            .
            .
      CALL EVWAIT(COMPLETE)           ! wait for task B to complete one task
      CALL EVCLEAR(COMPLETE)
C           Test Process: convergence
            .
            .
            .
            .
C        if the test shows no more processing necessary DO=False
      GO TO 10                        ! task must turn itself off when it is done
 20   CONTINUE
            .
            .
            .
            .
      END



C       Subroutine to be "fired up" as a task
      SUBROUTINE B( ,...)
      COMMON/MSG1/DO,A(128,128),READY,COMPLETE
      LOGICAL DO
 5    CALL EVWAIT(READY               ! wait to do next pass
      CALL EVCLEAR (READY)            ! once through at a time
      IF(DO.NE.TRUE)STOP              ! task is deactivated by stop
C           Process B: black portion of matrix A
            .
            .
            .
      CALL EVPOST(COMPLETE)           ! signal "ready for test"
      GO TO 5
      END
```

To determine if an event has been posted the programmer needs the following function:

```
EVTEST ( integer-variable )
```

The contents of the integer-variable must be the value generated by the multi-tasking support library on a call to EVASGN.

This is a logical function and the programmer will also need a LOGICAL EVTEST type declaration for it in the program.

This function will return a logical "TRUE" if the event has been posted and a logical "FALSE" if the event has never been posted or has been cleared.

(NOTE: This is unlike the LOCKTEST function in that it does not post the event.)

To release an event identifier that was defined with an EVASGN the programmer needs the following statement:

```
CALL EVREL ( integer-variable )
```

The contents of the integer-variable must be the value generated by the multi-tasking support library on a call to EVASGN.

The function of this routine is to detect erroneous uses of the event after the region where the program has planned for it. Any reference to this identifier while it remains unassigned is an error, although the event may be used after another call to EVASGN.

SECTION 5

MULTITASKING: HOW TO DO IT

OBJECTIVE: AT THE END OF THE COURSE THE LEARNER IS ABLE TO CONVERT AN OPTIMIZED FORTRAN PROGRAM RUNNING ON A SINGLE PROCESSOR OF AN X-MP TO DO MULTITASKING UTILIZING A SYSTEMATIC APPROACH TO THE PROBLEM.

WHY DO WE DO IT?

MULTIPROCESSING CAN BE USED TO IMPROVE PERFORMANCE WITH RESPECT TO TIME.

WHEN IS IT APPROPRIATE?

THE CODE MUST PRODUCE THE RIGHT RESULTS AND BE CLEAN (IT HAS NO DEADCODE, NO UNINITIALIZED VARIABLES).

CODE THAT HAS BEEN THROUGHLY OPTIMIZED USING OTHER TECHNIQUES SUCH AS VECTORIZATION, IMPROVEMENT OF SCALAR PERFORMANCE AND I/O IMPROVEMENTS BUT STILL IS TOO SLOW OR TOO RESOURCE CONSUMING.

BAD CODE MAKES THE HARD JOB OF CONVERTING SINGLE PROCESSOR CODE TO MULTIPROCESSOR CODE ALMOST IMPOSSIBLE.

OTHER SPEEDUP TECHNIQUES: ARE EASIER THAN MULTIPROCESSING: THEY MAY PRODUCE A GREATER IMPROVEMENT (VECTORIZATION CAN GIVE AN IMPROVEMENT FACTOR OF 20 AS COMPARED TO IMPROVEMENT FACTOR 2 FOR 2 PROCESSOR MULTIPROCESSING).

ANALYSIS OF THE CODE FOR MULTIPROCESS RELIES ON RUN TIMES. THEREFORE OTHER OPTIMIZATIONS SHOULD BE DONE FIRST.

HOW SHOULD IT BE DONE?

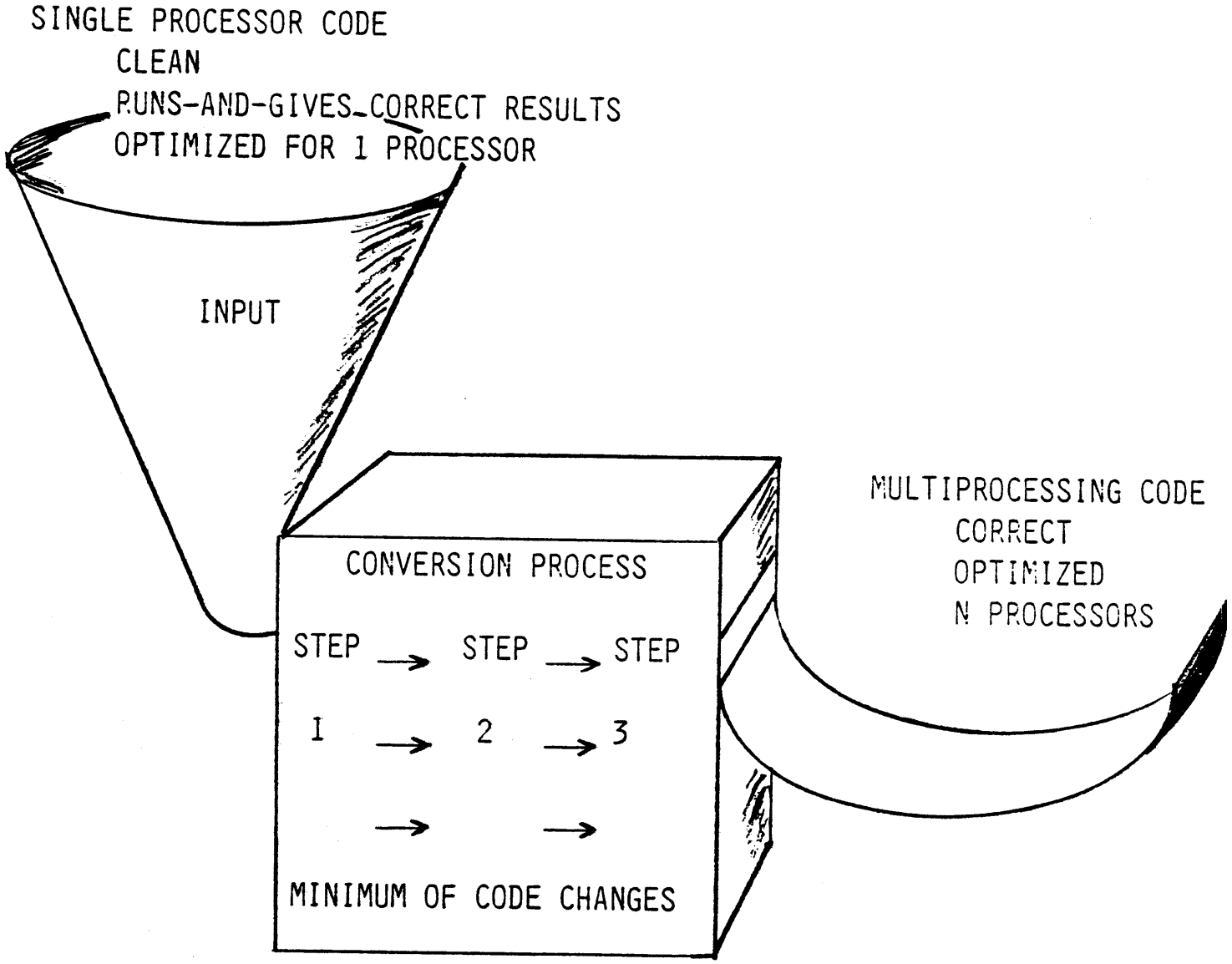
THE EASIEST WAY POSSIBLE; WITH A MINIMUM OF CODING CHANGES.

WHAT SHOULD IT LOOK LIKE WHEN WE'RE DONE?

MULTIPROCESSING CODE THAT IS INDEPENDENT OF THE # OF PROCESSORS (WE CAN'T PREDICT HOW MANY WILL BE AVAILABLE TODAY MUCH LESS TOMORROW).

IT IS THE FASTEST CODE POSSIBLE. WE DIDN'T GIVE UP MORE IN VECTORIZATION AND ADDED OVERHEAD THAN WE GAINED IN MULTIPROCESSING.

MULTIPROCESSING FOR A SPEEDUP



STEP 1: IDENTIFY PARTS OF THE WORK THAT COULD BE DONE IN PARALLEL.

STEP 2: VERIFY THAT THE PARTS CAN BE MADE INTO TASK; THAT SHARED AND LOCAL DATA CAN BE HANDLED CORRECTLY; AND THAT SYNCHRONIZATION ALWAYS WILL WORK.

STEP 3: WRITE THE MULTITASKING CODE.

STEP 1

IDENTIFY PARTS OF THE WORK THAT CAN BE DONE IN PARALLEL.

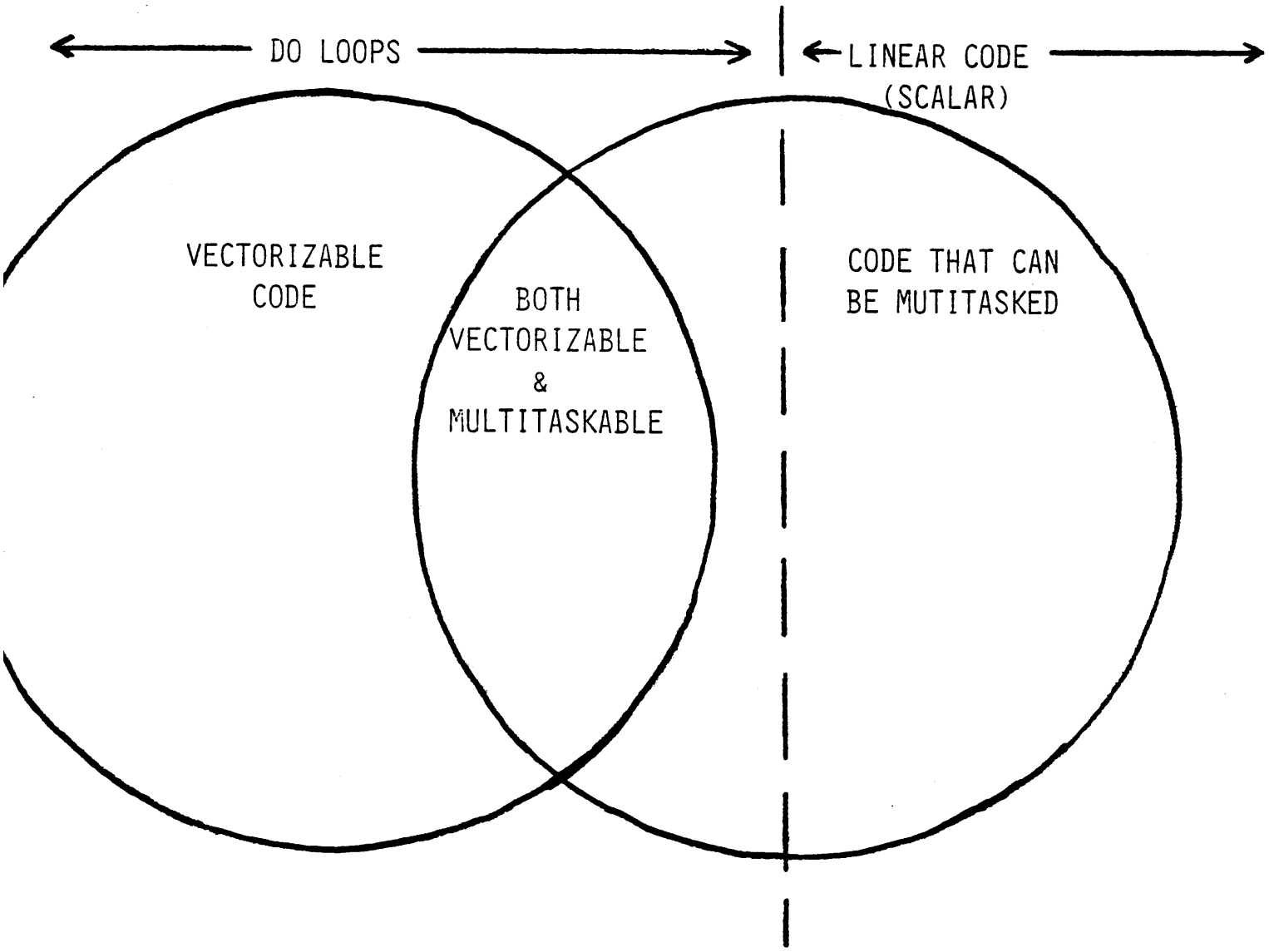
PARTS OF THE WORK MAY MEAN DIFFERENT SEGMENTS OF CODE OR ITERATION OF THE SAME SEGMENT.

VECTORIZATION IS A FORM OF PARALLEL PROCESSING THAT ALLOWS A NUMBER OF ITERATIONS OF THE SAME SEGMENT TO BE DONE IN PARALLEL.

SOME CODE CAN BE VECTORIZED AND MULTITASKED.

SOME CODE CAN BE VECTORIZED BUT NOT MULTITASKED.

SOME CODE CANNOT BE VECTORIZED BUT CAN BE MULTITASKED.

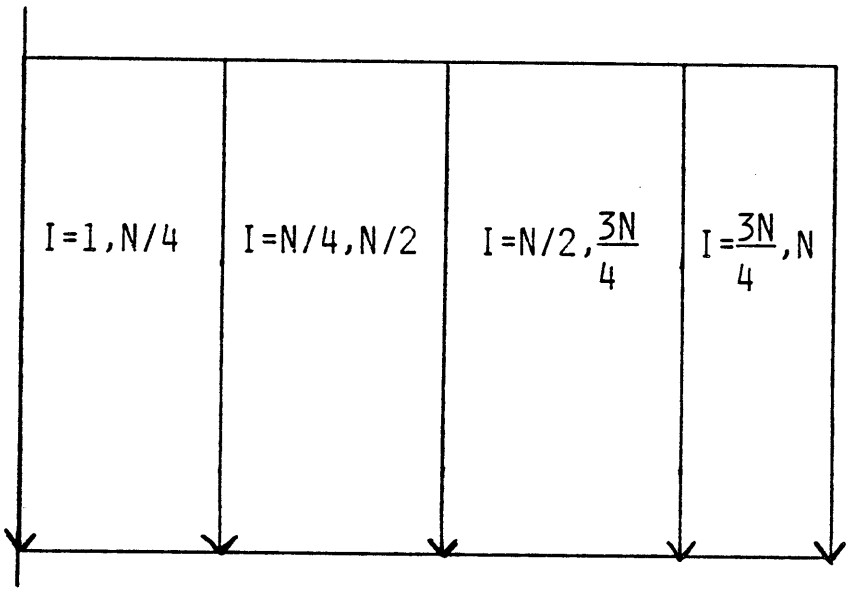


WE DO NOT WANT TO DIMINISH VECTORIZATION IN ORDER TO MULTITASK.

PARALLEL CODE

VECTORIZABLE & MULTITASKABLE

```
      DO 5  I=1,N
       A(I)=B(I)              !WILL VECTORIZE
5     CONTINUE
```

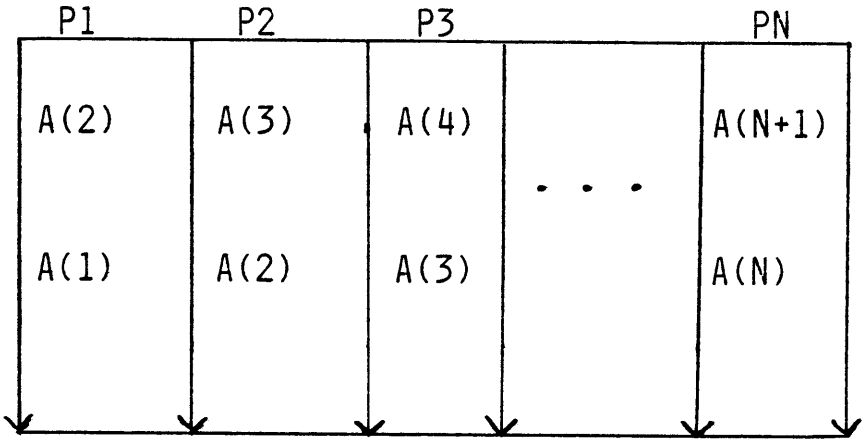


(SECTIONED DOALL)

PARALLEL CODE

VECTORIZABLE BUT NOT MULTITASKABLE

```
      DO  35  I=1,N,1
      A(I)=A(I+1)
35    CONTINUE
```



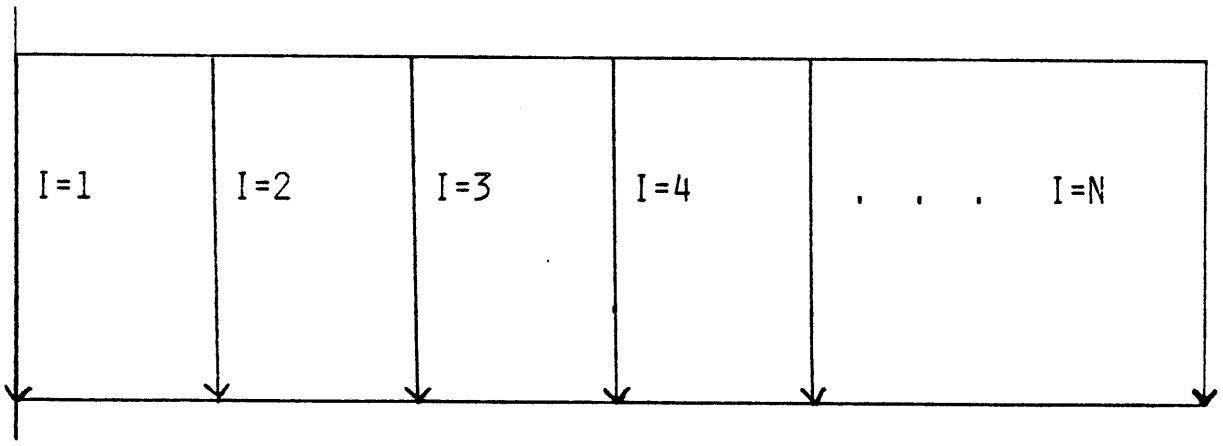
P3 MAY FINISH BEFORE P2 GETS STARTED.

RESULTS ARE UNPREDICTABLE.

PARALLEL CODE

NON-VECTORIZABLE BUT MULTITASKABLE

```
      DO  24  I=1,N
      DO  15  J=1,I
       IF (J.EQ.1)THEN          !THIS WON'T VECTORIZE
          A(J,I)=0
       ELSE
          A(J,I)=A(J-1,I)+1
       ENDIF
15    CONTINUE
25    CONTINUE
```



"PARTS OF THE WORKS" = CODE SEGMENTS

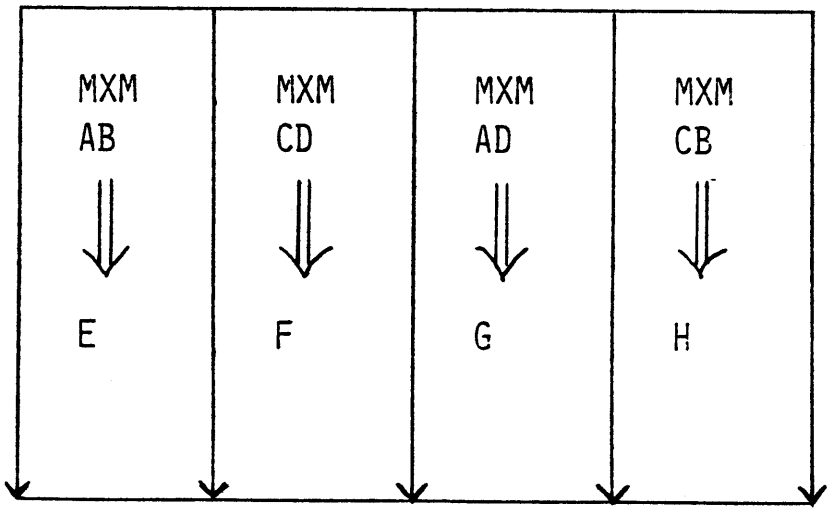
PARALLELISM CAN BE FOUND AT LEVELS HIGHER THAN ITERATIVE LOOPS.

CODE SEGMENTS COULD BE SUBROUTINES THAT ARE INDEPENDENTLY CALLED.
(IF THEIR CALLING SEQUENCE CAN BE CHANGED WITHOUT AFFECTING THE RESULTS THEN THEY CAN PROBABLY BE DONE IN PARALLEL).

CALLS CAN BE TO THE SAME SUBROUTINE OR TO DIFFERENT SUBROUTINES.

CRITICAL REGIONS OF CODE MUST BE HANDLED APPROPRIATELY.

```
PARALLEL CODE

MULTITASKABLE

PROGRAM CALLS

DIMENSION A(100,100),B(100,100,C(100,100),D(100,100)
READ*,A,B,C,D

CALL MXM ( A,100,B,100,E,100)

CALL MXM ( C,100,D,100,F,100)

CALL MXM ( A,100,D,100,G,100)

CALL MXM ( B,100,c,100,H,100)

WRITE*,E,F,G,H
```

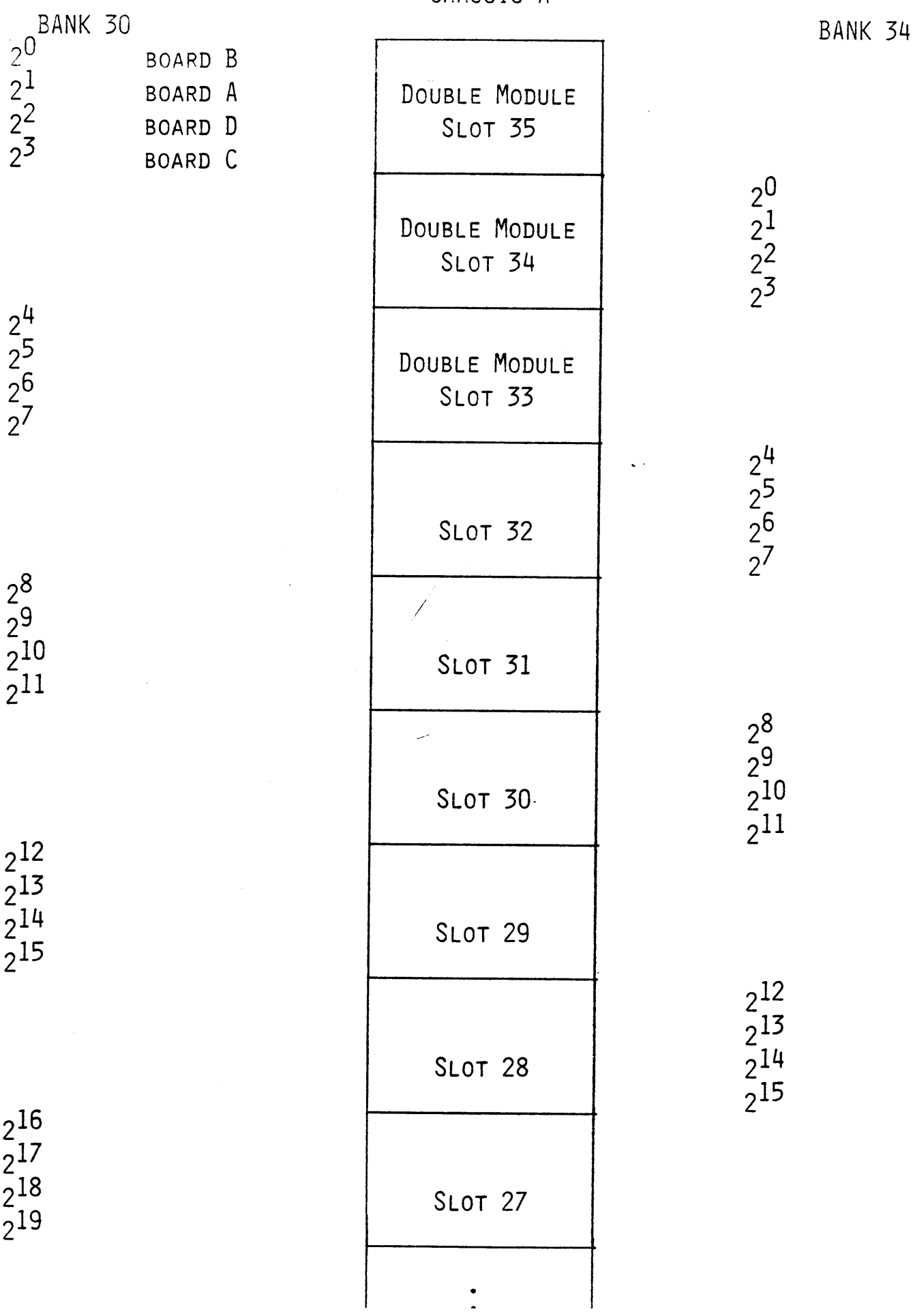


(COBEGIN)

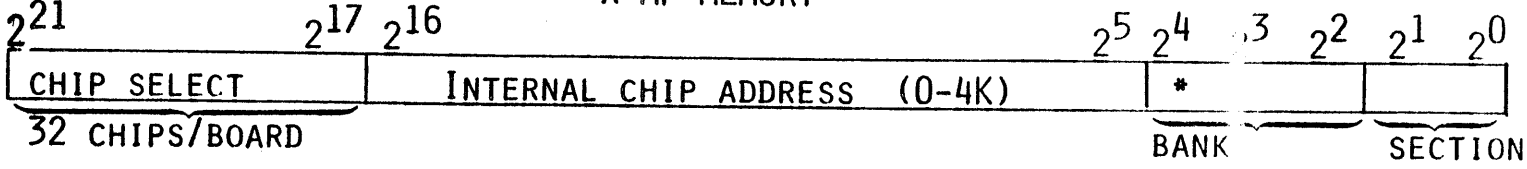
APPENDIX A

HARDWARE

X-MP MEMORY


CHASSIS A
BANK 30
BANK 34
2^0
2^1
2^2
2^3
BOARD B
BOARD A
BOARD D
BOARD C
DOUBLE MODULE SLOT 35
DOUBLE MODULE SLOT 34
DOUBLE MODULE SLOT 33
SLOT 32
SLOT 31
SLOT 30
SLOT 29
SLOT 28
SLOT 27
2^4
2^5
2^6
2^7
2^8
2^9
2^{10}
2^{11}
2^{12}
2^{13}
2^{14}
2^{15}
2^{16}
2^{17}
2^{18}
2^{19}

X-MP MEMORY



2^0

CHASSIS A	CHASSIS B	CHASSIS C	CHASSIS D
BANK 30 BANK 34	BANK 20 BANK 24	BANK 10 BANK 14	BANK 00 BANK 04

2^{63} CB0 ⋮ CB7

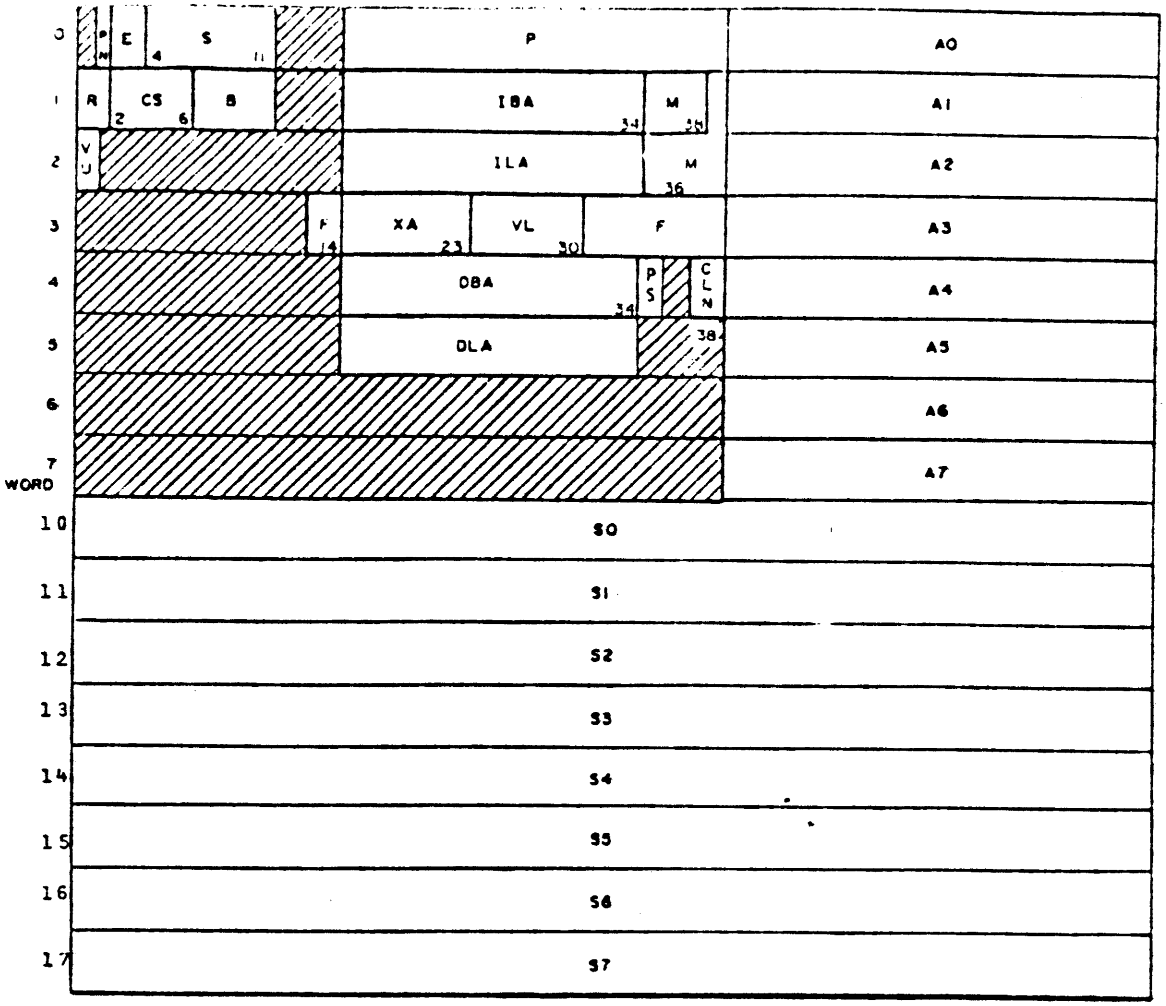
CHASSIS M	CHASSIS N	CHASSIS O	CHASSIS P
BANK 31 BANK 35	BANK 21 BANK 25	BANK 11 BANK 15	BANK 01 BANK 05

CB7 ⋮ CB0 2^{63}

2^0

CHASSIS I	CHASSIS J	CHASSIS K	CHASSIS L
BANK 02 BANK 06	BANK 12 BANK 16	BANK 22 BANK 26	BANK 32 BANK 36

CHASSIS U	CHASSIS V	CHASSIS W	CHASSIS X
BANK 03 BANK 07	BANK 13 BANK 17	BANK 23 BANK 27	BANK 33 BANK 37



READ OUT AREA

WORD	BIT	FIELD
0	2^{62}	PN-Processor Number
0	2^{61}-2^{60}	E- Error Type
0	2^{59}-2^{52}	S- Syndrome Bits
1	2^{63}-2^{62}	R-Read Mode
1	2^{61}-2^{57}	CS-Chip Select Bits
1	2^{56}-2^{52}	B- Bank Address
2	2^{63}	VU - Vectors Used

STATUS AREA

WORD	BIT	FIELD
0	2^{24}-2^{47}	P - Program Add. Reg.
1	2^{29}-2^{47}	IBA-INST. BASE ADD.
2	2^{29}-2^{47}	ILA-INST. LIMIT ADD.
3	2^{40}-2^{47}	XA-EXCH. ADD.
3	2^{33}-2^{39}	VL-Vector Length
4	2^{29}-2^{47}	DBA-Data Base Address
5	2^{29}-2^{47}	DLA-Data Limit Address

17 bits

WORD	BIT	FIELD
1&2	2^{24}-2^{28}	M-Mode Reg.
1	2^{24}	IMM-Interrupt Monitor Mode
1	2^{25}	SEI-Select for Ext. Interr
1	2^{26}	BDM-Bidirect. Mem. Access
1	2^{27}	FPS-Fl. Pt. Error Status
1	2^{28}	WS-Waiting on Semaphore
2	2^{24}	MM-Monitor Mode
2	2^{25}	IUM-Interr. on Uncorr. Mem
2	2^{26}	IFP-Interr. on F. Pt. Erro
2	2^{27}	ICM-Interr. on Corr. Mem.
2	2^{28}	IOR-Interr. on Op. Range E
FLAGS		
3	2^{49}	ICP-Interr. from Int. CPU
"	2^{48}	DL-Deadlock
"	2^{32}	PCI-Prog. Clk
"	2^{31}	MCU-MCU Interr. (MIOP)
"	2^{30}	FPE-Fl. Pt. Error
"	2^{29}	ORE-Op. Range Err
"	2^{28}	PRE-Prog. Range Error
"	2^{27}	ME-Memory Error
"	2^{26}	IOI-I/O Interrupt
"	2^{25}	EEX-Error Exit (000)
"	2^{24}	NEX-Normal Exit (004)
4	2^{28}	PS-Program State
4	2^{24}-2^{25}	CLN- Cluster Number Regist

No internal I/O address)

X-MP HARDWARE

Logic is Emitter-Coupled Logic (ECL) as in CRAY 1-S.

16 pin flat packs (chip types are used as in CRAY 1-S, but :

X-MP uses gate arrays (many gates)
1-S uses 5/4 and gates (2 gates).

As in the CRAY-1S:

Signals (outputs) are produced with their complements.
60Ω resistors are used to "tie down" each output.
Testpoints are the same.

Boolean equations are written for the normal outputs of both 1-S and X-MP, but for the X-MP there are 2 types of boolean.

Macro Boolean: Shorthand for trouble shooting using test points.
Doesn't have chip types or identifiers

Rigid Boolean: As for the CRAY 1-s.
Used for scoping modules once they are extended.

BASIC TIMING

9.5 nsec normal clock
  ≤9.0 nsec for logic and foil runs
  ≥0.5 nsec for synchronization (latching)

Clock periods (cp's) are devided into 19 time segments (TS) numbered 0 to 18 (actually #17 and #18 aren't used).

Latch segments are TS 15/0 or TS 16/0.

Boolean logic falls between TS0 and TS15.

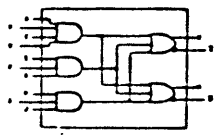
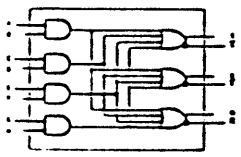
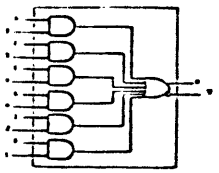
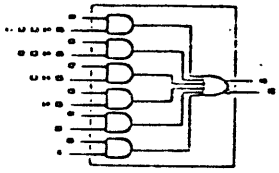
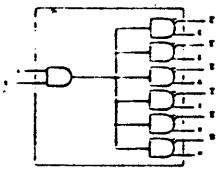
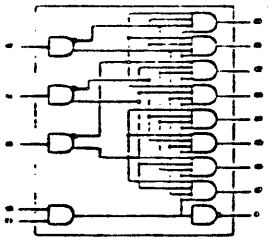
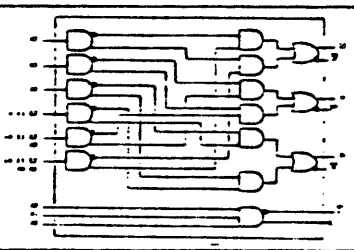
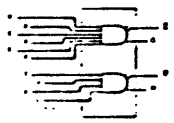
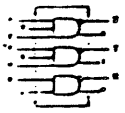
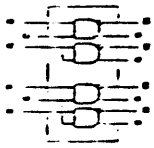
1 TS = 0.5 nsec = 2.5" of foil
= 4" of wire (measured connector to connector).

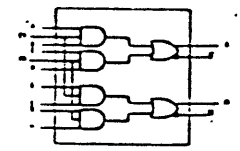
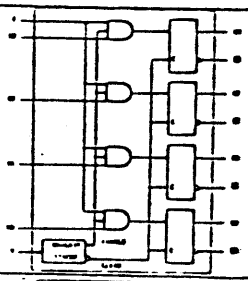
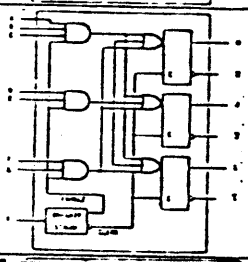
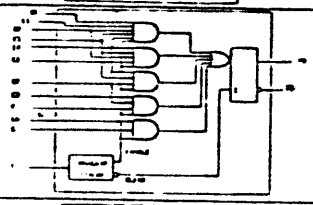
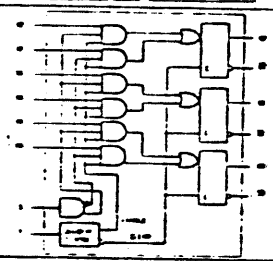
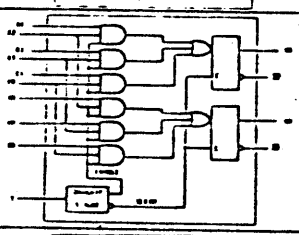
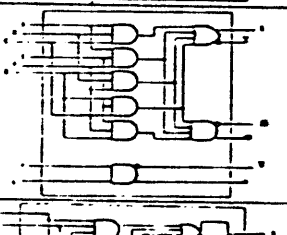
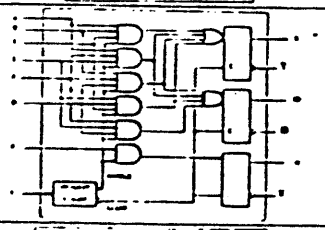
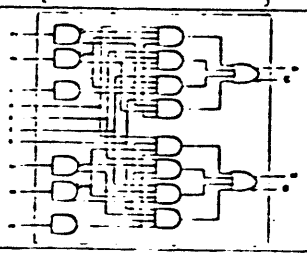
(The speed of light is ~ 11.8 inches/nsec).

Gate arrays are divided in 3 levels:

Level 1 = 2TS = 1 nsec
Level 2 = 3TS = 1.5 nsec
Level 3 = 4TS = 2.0 nsec

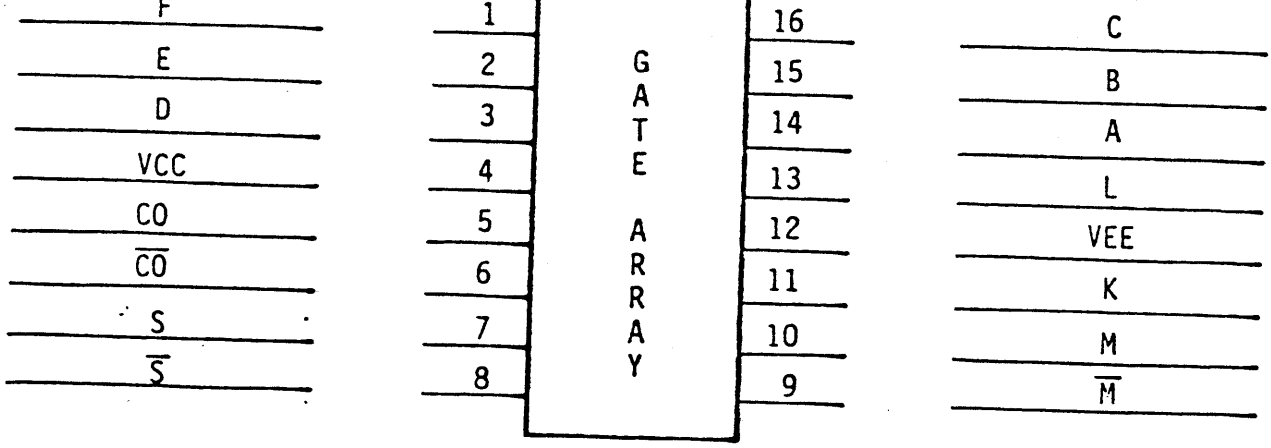
TYPE	LEVEL
A-F	2
G	3
H	1
I	1
J	1
K	2
L	2
M	2
N	2
O	2 & 3
P	2
Q	2
T	2
U	2 & 3
W	2
Y	2

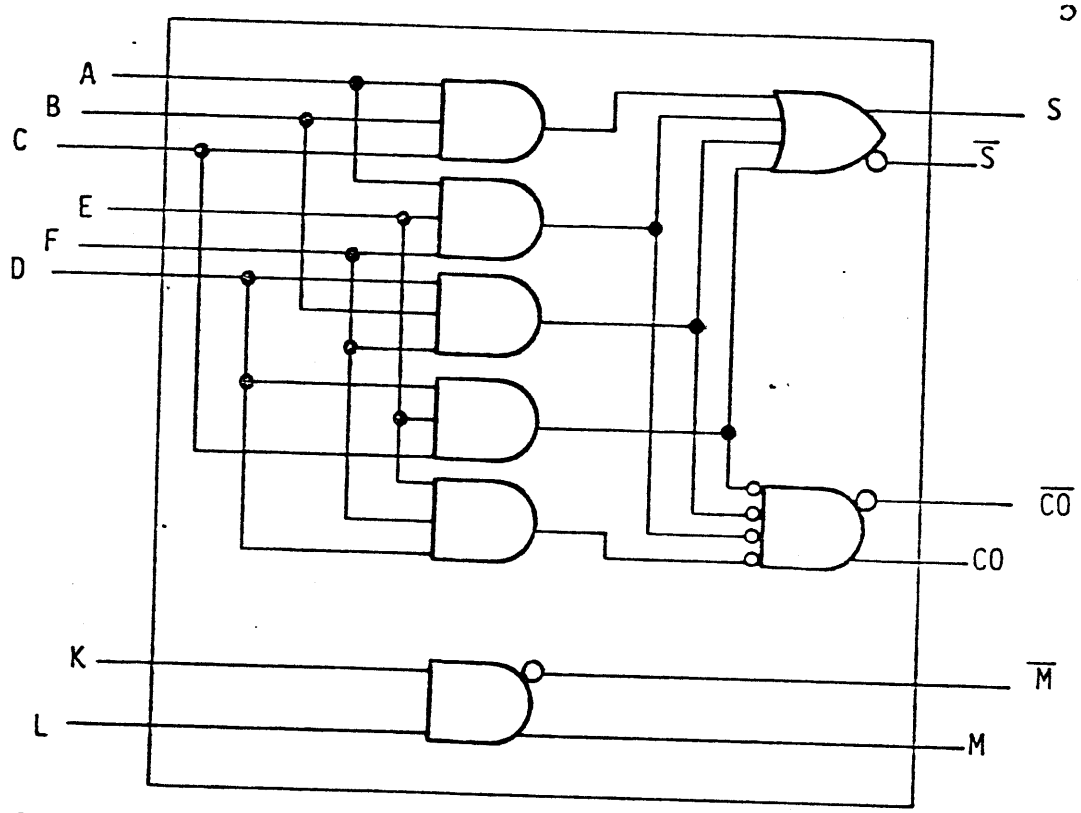
EQUATIONS	SCHEMATIC	POWER
K, M, K, M = A B C D + E F G + H I J 5, 7, 5, 8 = 13 14 15 16 + 9 10 11 + 1 2 3 Macro Definition: K, M = A B C D + E F G + H I J		VOLT MW 5.2 .267 2 int .054 2 ter .054 TOTAL .375
I, K, M, I, K, M = A B + C D + E F + G H 9, 8, 5, 10, 7, 6 = 11 13 + 14 15 + 16 1 + 2 3 Macro Definition: I, K, M = A B + C D + E F + G H		VOLT MW 5.2 .413 2 int .081 2 ter .081 TOTAL .575
M M = A B + C D + E F + G H + I J + K L 3 7 = 5 6 + 9 10 + 11 13 + 14 15 + 16 1 + 2 3 Macro Definition: M + A B + C D + E F + G H + I J + K L		VOLT MW 5.2 .359 2 int .027 2 ter .027 TOTAL .413
P6 = C0 E1 E2 E3 E4 E5 + C1 E2 E3 E4 E5 + C2 E3 E4 E5 + C3 E4 E5 + C4 E5 + C5 G 7 = 9 10 15 11 14 13 + 16 15 11 14 13 + 1 11 14 13 + 5 14 13 + 6 13 + 2 3 acro Definition: P6 = PCARY(C5-0, E5-1) = C5 C4 C3 C2 C1 C0 E5 E4 E3 E2 E1 # = 2 6 5 1 16 9 13 14 11 15 10 3		VOLT MW 5.2 .425 2 int .027 2 ter .027 TOTAL .479
E, E, G, I, K, M, C, E, G, I, K, M = A B 9, 7, 10, 13, 16, 2, 5, 8, 9, 14, 15, 1 = 3 11 Macro Definition: E, E, G, I, K, M = A B		VOLT MW 5.2 .393 2 int .162 2 ter .162 TOTAL .717
0 = A2 A1 A0 E1 E0 Q1 = A2 A1 A0 E1 E0 Q2 = A2 A1 A0 E1 E0 Q3 = A2 A1 A0 E1 E0 4 = A2 A1 A0 E1 E0 Q5 = A2 A1 A0 E1 E0 Q6 = A2 A1 A0 E1 E0 Q7 = A2 A1 A0 E1 E0 = E1 E0 = 13 11 Macro Definition: 7-0 = DCD(A2-0)/E1 E0 Q7 Q6 Q5 Q4 Q3 Q2 Q1 Q0 = A2 A1 A0 E1 E0 = E1 E0 10 9 8 7 1 16 15 14 2 3 5 13 11		VOLT MW 5.2 .629 2 int .139 2 ter .195 TOTAL .963
2 Q2 = A2(E0 E1 E2 A0 A1) + A2(E0 E1 E2 A0 A1) Q0 Q0 = A0(E0 E1 E2) + A0(E0 E1 E2) 10 = 11 1 2 3 14 13 + 11 1 2 3 14 13 5 6 = 14 1 2 3 + 14 1 2 3 1 Q1 = A1(E0 E1 E2 A0) + A1(E0 E1 E2 A0) C C = A2 A1 A0 7 = 13 1 2 3 14 + 13 1 2 3 14 15 16 = 11 13 14 acro Definition: 2-0 = SUM(A2-0, 1)/E2 E1 E0 Q2 Q1 Q0 = A2 A1 A0 E2 E1 E0 C = A2 A1 A0 = A2 A1 A0 10 7 6 = 11 13 14 3 2 1 16 = 11 13 14		VOLT MW 5.2 .788 2 int .108 2 ter .108 TOTAL 1.004
G = A B C D E F M M = I J K L 8 = 6 5 3 2 1 16 10 9 = 11 13 14 15 Macro Definition: = A B C D E F M = I J K L		VOLT MW 5.2 .140 2 int .054 2 ter .054 TOTAL .248
D = A B C I I = F G H M M = K L 10 = 11 13 14 5 6 = 3 2 1 7 8 = 15 16 acro Definition: = A B C I = F G H M = K L		VOLT MW 5.2 .176 2 int .081 2 ter .081 TOTAL .338
Q3 = D3 E1 Q2 Q2 = D2 E1 Q1 Q1 = D1 E0 Q0 Q0 = D0 E0 8 = 6 5 10 9 = 11 5 15 16 = 14 13 2 1 = 3 13 cro Definition: = D0 E0 Q1 = D1 E0 Q2 = D2 E1 Q3 = D3 E1		VOLT MW 5.2 .279 2 int .108 2 ter .108 TOTAL .495

EQUATIONS	SCHEMATIC	POWER
$\overline{K}$ K = A B E1 + C D EO $\overline{M}$ M = E F E1 + G H EO 10 9 = 13 14 11 + 15 16 3 8 7 = 1 2 11 + 5 6 3 Macro Definition: K = A B E1 + C D EO M = E F E1 + G H EO		VOLT MW 5.2 .377 2 int .054 2 ter .054 TOTAL .485
$\overline{Q0}$ Q0 = D0 E; T $\overline{Q1}$ Q1 = D1 E; T $\overline{Q2}$ Q2 = D2 E; T $\overline{Q3}$ Q3 = D3 E; T 2 1 = 3 5;13 15 16 = 14 5;13 10 9 = 11 5;13 7 8 = 6 5;13 Macro Definition: Q0 = D0 E;T Q1 = D1 E;T Q2 = D2 E;T Q3 = D3 E;T		VOLT MW 5.2 .881 2 int .108 2 ter .108 TOTAL 1.097
$\overline{H}$, $\overline{J}$, $\overline{L}$, H, J, L = A B C + D E + F G; T 5, 8, 9, 6, 7, 10 = 15 16 1 + 11 14 + 2 3;13 Macro Definition: H, J, L = A B C + D E + F G;T		VOLT MW 5.2 .549 2 int .081 2 ter .081 TOTAL .711
$\overline{P5}$ P5 = C0 E1 E2 E3 E4 + C1 E2 E3 E4 + C2 E3 E4 + C3 E4 F + C4 G; T 9 10 = 8 7 6 1 11 + 5 6 1 11 + 16 1 11 + 15 11 14 + 3 2;13 Macro Definition: P5 = PCARY(C4-0, E4-1);T P5 = C4 C3 C2 C1 C0 E4 E3 E2 E1 # #; T 10 = 3 15 16 5 8 11 1 6 7 14 2;13		VOLT MW 5.2 .561 2 int .027 2 ter .027 TOTAL .615
$\overline{Q0}$ Q0 = D0 S + B0 $\overline{S}$;T $\overline{Q1}$ Q1 = D1 S + B1 $\overline{S}$; T $\overline{Q2}$ Q2 = D2 S + B2 $\overline{S}$; T 6 7 = 2 5 + 3 5;13 9 10 = 11 5 + 8 5;13 16 15 = 14 5 + 1 5;13 Macro Definition: Q0 = MUX(D0 B0):DCD(S);T Q1 = MUX(D1 B1):DCD(S);T Q2 = MUX(D2 B2):DCD(S);T		VOLT MW 5.2 .888 2 int .081 2 ter .081 TOTAL 1.050
$\overline{Q0}$ Q0 = A0 E2 + B0 E1 + C0 E0; T $\overline{Q1}$ Q1 = A1 E2 + B1 E1 + C1 E0; T 6 5 = 7 10 + 8 11 + 9 14;13 3 2 = 15 10 + 16 11 + 1 14;13 Macro Definition: Q0 = A0 E2 + B0 E1 + C0 E0;T Q1 = A1 E2 + B1 E1 + C1 E0;T		VOLT MW 5.2 .725 2 int .054 2 ter .054 TOTAL .833
$\overline{S}$ S = A E F + D B F + D E C + A B C C0 $\overline{C0}$ = A E F + D B F + D E C + D E F 8 7 = 14 2 1 + 3 15 1 + 3 2 16 + 14 15 16 5 6 = 14 2 1 + 3 15 1 + 3 2 16 + 3 2 1 $\overline{M}$ M = K L 9 10 = 11 13 Macro Definition: S = SUM(A,B,C) $\overline{C0}$ = CARY(A,B,C) M = K L Note: This is true only for D = $\overline{A}$ E = $\overline{B}$ F = $\overline{C}$		VOLT MW 5.2 .387 2 int .081 2 ter .081 TOTAL .549
$\overline{S}$ S = A E F + D B F + D E C + A B C; T C0 $\overline{C0}$ = A E F + D B F + D E C + D E F; T 1 2 = 8 14 6 + 5 7 6 + 5 14 3 + 8 7 3;13 16 15 = 8 14 6 + 5 7 6 + 5 14 3 + 5 14 6;13 $\overline{M}$ M = K 10 9 = 11 Macro Definition: S = SUM(A,B,C);T $\overline{C0}$ = CARY(A,B,C);T M = K;T Note: This is true only if D = $\overline{A}$ E = $\overline{B}$ F = $\overline{C}$		VOLT MW 5.2 .845 2 int .081 2 ter .081 TOTAL 1.007
$\overline{Q0}$ Q0 = A $\overline{S1}$ $\overline{S0}$ E0 + B $\overline{S1}$ S0 E0 + C S1 $\overline{S0}$ E0 + D S1 S0 E0 9 10 = 1 14 15 11 + 16 14 15 11 + 13 14 15 11 + 5 14 15 11 $\overline{Q1}$ Q1 = A $\overline{P1}$ $\overline{P0}$ E1 + B $\overline{P1}$ P0 E1 + C P1 $\overline{P0}$ E1 + D P1 P0 E1 8 7 = 1 3 2 6 + 16 3 2 6 + 13 3 2 6 + 5 3 2 6 Macro Definition: Q0 = MUX(A B C D):DCD(S1 S0)/E0 Q1 = MUX(A B C D):DCD(P1 P0)/E1		VOLT MW 5.2 .500 2 int .054 2 ter .054 TOTAL .908

GATE ARRAY MACRO NAMES

GATE TYPE	NAME
A	Sum of Products (4,3,3) with Dual Fanout
B	Sum of Products (2,2,2,2) with Triple Fanout
C	Sum of Products (2,2,2,2,2,2)
D	Propagate Carry Generate
E	2-Input Gate with Hex Fanout
F	1-of-8 Decoder
G	+1 Adder
H	6/4 Gate
I	3/3/2 Gate
J	Dual 2-Input Buffer with Enable
K	Dual Sum of Products (3,3) with Common Enables
L	4-Bit Data Latch with Enable
M	Latched Sum of Products (3,2,2) with Triple Fanout
N	Latched Propagate Carry Generate
O	Latched Triple 2-to-1 MUX
P	Latched Dual 3-to-1 MUX
Q	3-Bit Adder
R	16x4 Register
S	4kx1 Memory
T	Latched 3-Bit Adder
U	Dual 4-to-1 MUX
W	Differential Input Gate with Hex Fanout
Y	Latched Sum of Products (2,2,2,2,1) with Dual Fanout





$S = A\ E\ F + D\ B\ F + D\ E\ C + A\ B\ C$ ...

$7 = 14\ 2\ 1 + 3\ 15\ 1 + 3\ 2\ 16 + 14\ 15\ 16$

$\overline{S} = \overline{A\ E\ F + D\ B\ F + D\ E\ C + A\ B\ C}$

$8 = 14\ 2\ 1 + 3\ 15\ 1 + 3\ 2\ 16 + 14\ 15\ 16$

$\overline{CO} = A\ E\ F + D\ B\ F + D\ E\ C + D\ E\ F$

$6 = 14\ 2\ 1 + 3\ 15\ 1 + 3\ 2\ 16 + 3\ 2\ 1$

$CO = (\overline{A\ E\ F})(\overline{D\ B\ F})(\overline{D\ E\ C})(\overline{D\ E\ F})$

$5 = (14\ 2\ 1)(3\ 15\ 1)(3\ 2\ 16)(3\ 2\ 1)$

$M = K\ L$

$10 = 11\ 13$

$\overline{M} = \overline{K\ L}$

$9 = 11\ 13$

Macro Definition:

S = SUM(A,B,C)

CO = CARY(A,B,C)

M = K L

NOTE: This is true only for

$D = \overline{A}$

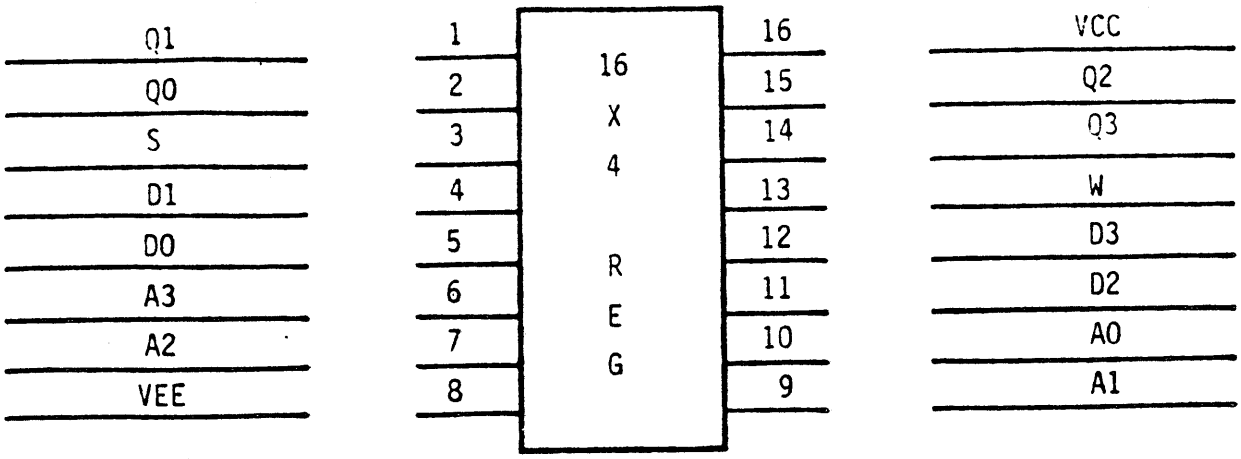
$E = \overline{B}$

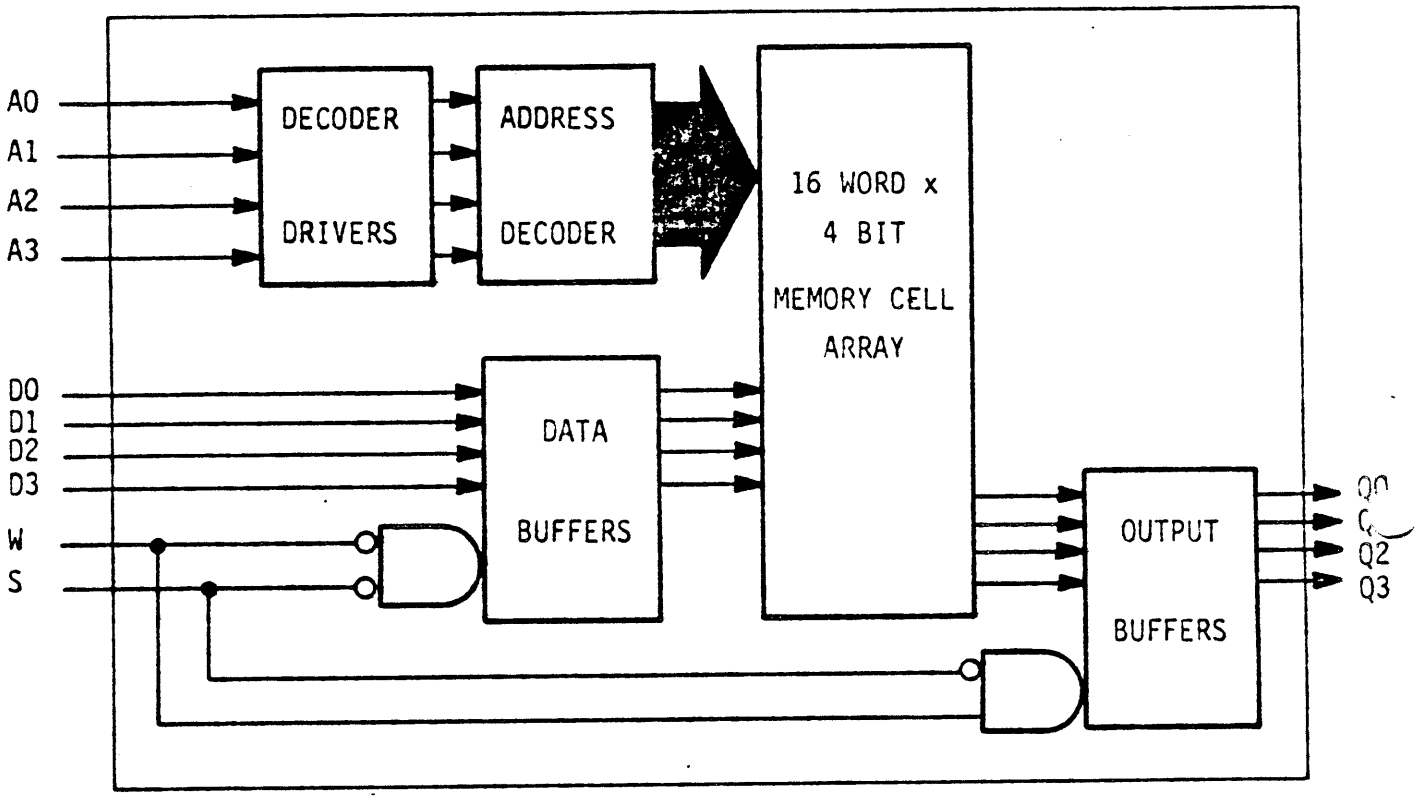
$F = \overline{C}$

Type: Q

Power Dissipation: -5.2 V. 387 MW. -2.0 V. 81 MW. Loading 468 MAT

Name: 3 Bit Adder





Macro Definition:

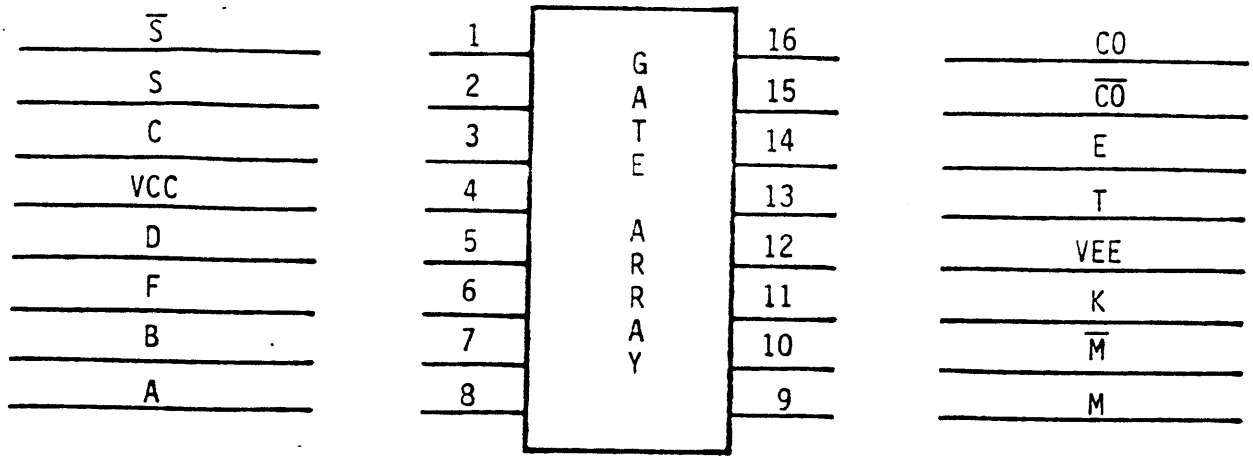
Q3 = D3:DCD(A3-0)/S + W

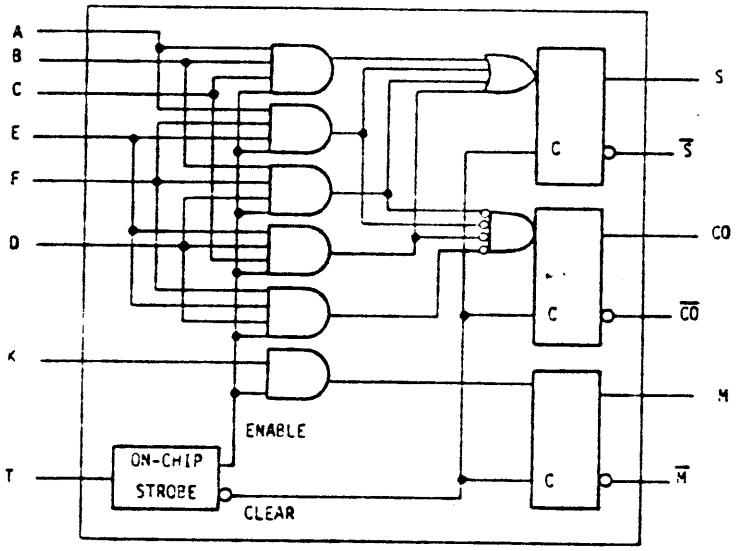
Q2 = D2:DCD(A3-0)/S + W

Q1 = D1:DCD(A3-0)/S + W

Q0 = D0:DCD(A3-0)/S + W

Type: R	Power Dissipation: $\frac{-5.2\ V.}{572\ MW.}$	$\frac{-5.2\ V.}{56\ MW.}$	$\frac{\text{Loading}}{628\ MWT}$
Name: 16 x 4 Register			





$S = A\,E\,F + D\,B\,F + D\,E\,C + A\,B\,C\ ;\ T$

$2 = 8\,14\,6 + 5\ 7\ 6 + 5\,14\,3 + 8\ 7\ 3\ ;\ 13$

$\overline{S} = \overline{A\,E\,F + D\,B\,F + D\,E\,C + A\,B\,C}\ ;\ T$

$1 = 8\,14\,6 + 5\ 7\ 6 + 5\,14\,3 + 8\ 7\ 3\ ;\ 13$

$\overline{CO} = A\,E\,F + D\,B\,F + D\,E\,C + D\,E\,F\ ;\ T$

$15 = 8\,14\,6 + 5\ 7\ 6 + 5\,14\,3 + 5\,14\,6\ ;\ 13$

$CO = (\overline{A\,E\,F})(\overline{D\,B\,F})(\overline{D\,E\,C})(\overline{D\,E\,F})\ ;\ T$

$16 = (8\,14\,6)(5\ 7\ 6)(5\,14\,3)(5\,14\,6)\ ;\ 13$

$M = K$ $\qquad$ $\overline{M} = \overline{K}$

$9 = 11$ $\qquad$ $10 = 11$

Macro Definition:

S = SUM(A, B, C);T

CO = CARY(A, B, C);T

M = K;T

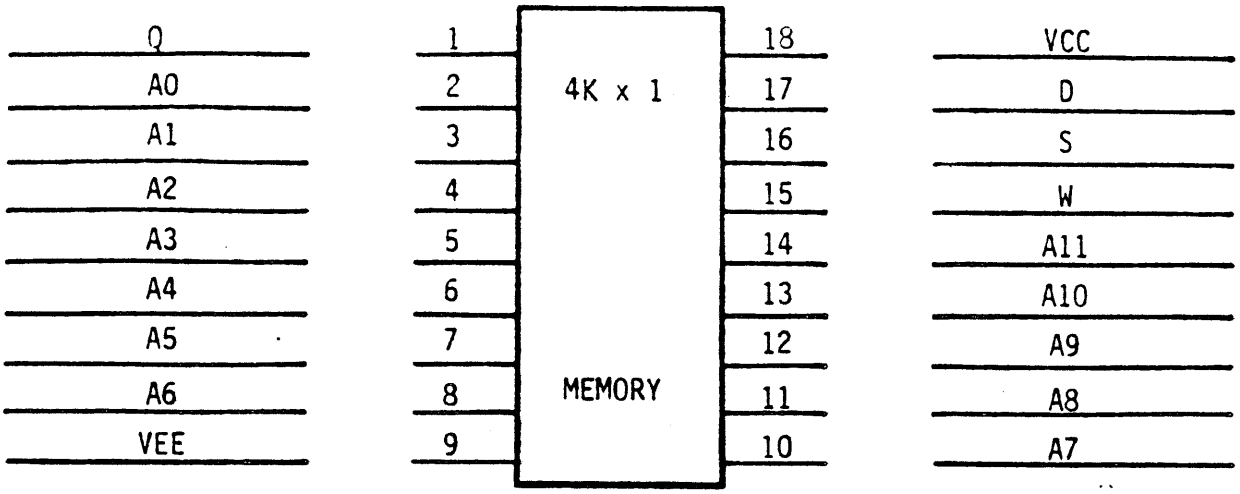
NOTE: This is true only if

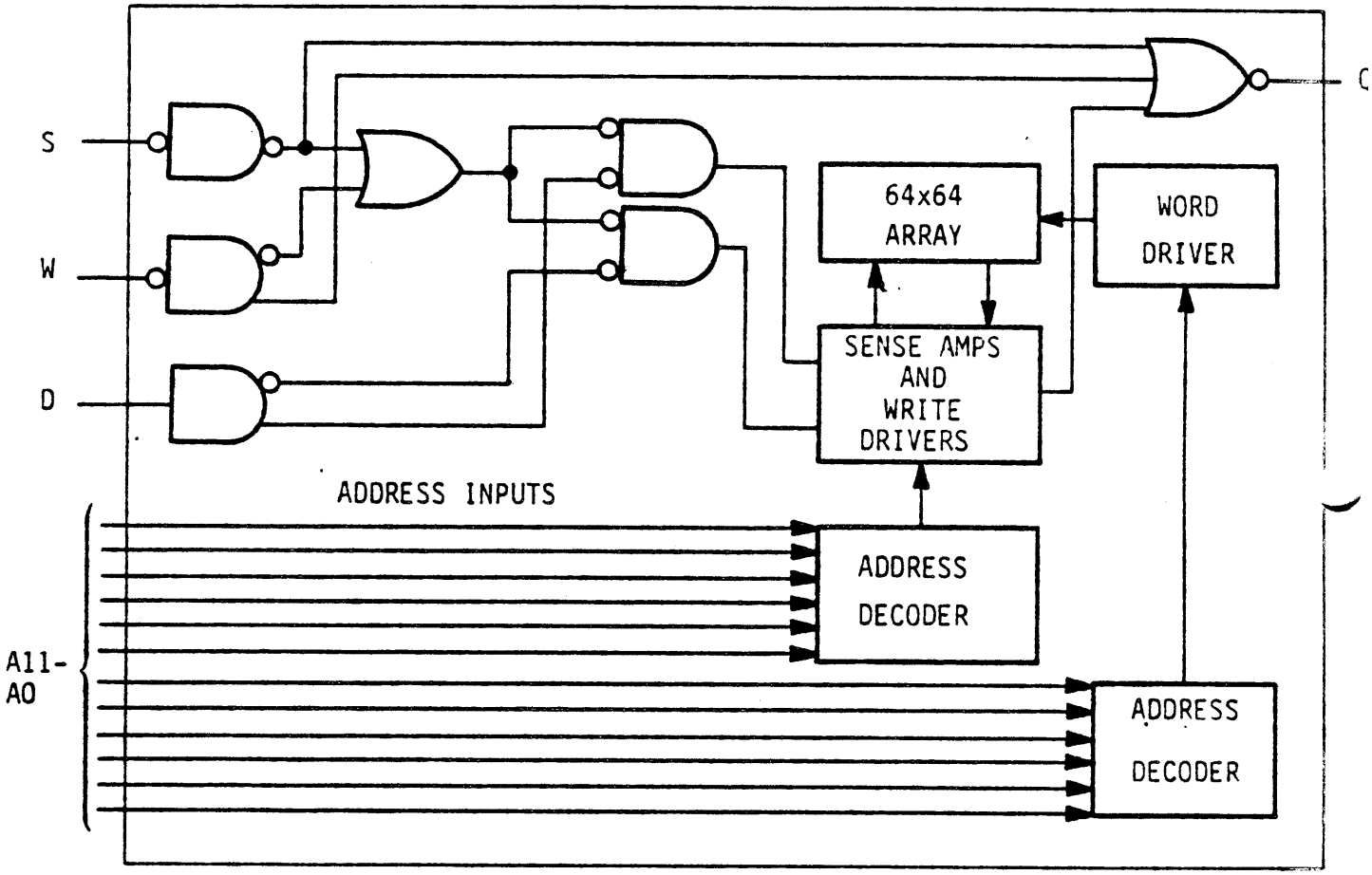
$D = \overline{A}$

$E = \overline{B}$

$F = \overline{C}$

Type: T	Power Dissipation:	-5.2 V. 845 MW.	-2.0 V. 81 MW.	Loading 926 MWT
Name: Latched 3-bit Adder				





Macro Definition:

Q = D : DCD(A11-0)/ S + W

Type: S	Power Dissipation:	-5.2 V. 780 MW.	-5.2 V. 14 MW.	Loading 794 MWI
Name: 4K x 1 MEMORY				

GATE ARRAY TERMINOLOGY

The term GATE ARRAY is used to describe a sixteen pin flatpak package which contains one or more logic circuits.

The term CIRCUIT is used to describe a logical function which is performed internal to the gate array. A circuit has one or more identical outputs which are logically formed from two or more inputs. Circuits are made up of "AND" and "OR" LOGIC GATES connected internal to the gate array.

Each circuit is correspondent to a BOOLEAN EQUATION defined in the gate array specification.

An input signal (pin) which is logically used in only one part of the same circuit is an INDEPENDENT INPUT. An input signal (pin) which is logically used in different parts of the same circuit or in different circuits is a COMMON INPUT. An independent input to a gate is equal to and interchangeable with any other independent in put to that gate. Any gate which has only independent inputs is an INDEPENDENT GATE. An independent gate is equal to and interchangeable with any other independent within the same circuit, provided the number of inputs needed to each gate are available.

The set of all gate array types which have common inputs, but contain interchangeable gates of interchangeable circuits is called a TYPE 2 GATE ARRAY. The interchange must also occur on all other circuits in the package which are affected by the common inputs. This type includes the types J,K,L and P.

The third set of gate array tyhpes is called TYPE 3 GATE ARRAY. This type of gate array has not only common inputs, but also distinct functions performed by each circuit. Hence, all the inputs and outputs have a specific ordering relation and format. Some interchangeability of inputs is possible as long as specific relationships are maintained.,

Exception: In the case of type R and S, the address inputs need <u>not</u> follow any specific <u>order</u> <u>or</u> have any specific <u>polarity</u>.

Type F - the <u>polarity</u> of the inputs A2-0 may be reversed as long as the <u>order</u> of the inputs Q7-0 are reversed.

Type Q and T - the order of the input pairs A,$\overline{A}$ and B,$\overline{B}$ and C,$\overline{C}$ is not important as long as the pair relationship is maintained.

Type O,U - the <u>polarity</u> of the DCD terms may be reversed as long as the <u>order</u> of the MUX terms is reversed.

This third set of gate array types includes types D,F,G,N,O,Q,R,S,T, and U.

MEMORY (MODULE LEVEL)

2 or 4 million words
    4K chips
all X's will be 12 column machines

To access 1 memory word
    18 modules must be accessed
    4 bits/module, 1 bit per board (S has 1 bit/module)
    Check bits still in the middle

Ports	Instruction
A	034, 176
B	036, 176
C	035, 037, 177, 100-137
I/O	

B,T,V look for a free port

Scalar reference act like in 1S (ranks are used)

QA3 modules handle the 4 sections for both CPU's
    Section 0 and 1 in chassis G (top to bottom ordering)
    Section 2 and 3 in chassis S

    Checks section conflicts for same CPU
        Bank conflicts overall

    Holds bank bits for 3 CP's if needed
    If both CPU wants same bank, priority decides which gets in
        (priority switches back & forth every 4 CP's)

Section conflicts from the same CPU are resolved by looking at:
    a) The increment value (if odd it's given priority), then
    b) If both even or both odd, the one that issued first goes first.

A signal is sent back to the 3AA or the 3AB module to say don't increment and send the same address again.

The 3AF module controls both FETCH & EXCHANGE.
    These operations will be given priority over everything, taking over completely and making everything busy.

A FETCH will wait until memory has quieted down (i.e., for No bank conflicts, other references are held up).
On Exchange memory must be quiet, all references have to complete first.

16 bank machine, fetch is slower

Can use innermost or outermost 2 columns referred to as upper and lower.

inside C & D chassis

Outside A & B chassis

(If interupt when operand range error is disable, the instruction will still not execute.)

APPENDIX B

CAL
SYMBOLIC MACHINE INSTRUCTIONS
FOR THE X-MP

INSTRUCTION SUMMARY

	CRAY-1	CAL	PAGE	UNIT	DESCRIPTION
→	000000	ERR	5-7	-	Error exit
	††0010*jk*	CA,A*j* A*k*	5-8	-	Set the channel (A*j*) current address to (A*k*) and begin the I/O sequence
	††0011*jk*	CL,A*j* A*k*	5-8	-	Set the channel (A*j*) limit address to (A*k*)
	††0012*j*0	CI,A*j*	5-8	-	Clear channel (A*j*) interrupt flag; clear device master-clear (output channel).
★	††0012*j*1	MC,A*j*	5-8	-	Clear channel (A*j*) interrupt flag; set device master-clear (output channel); clear device ready-held (input channel).
	††0013*j*0	XA A*j*	5-8	-	Enter XA register with (A*j*)
	††0014*j*0	RT S*j*	5-10	-	Enter RTC register with (S*j*)
★	††001401	IP 1	5-10	-	Set interprocessor interrupt
★	††001402	IP 0	5-10	-	Clear interprocessor interrupt
★	††001403	CLN 0	5-10	-	Enter CLN register with 0
★	††001413	CLN 1	5-10	-	Enter CLN register with 1
★	††001423	CLN 2	5-10	-	Enter CLN register with 2
★	††001433	CLN 3	5-10	-	Enter CLN register with 3
	††0014*j*4	PCI S*j*	5-10	-	Enter II register with (S*j*)
	††001405	CCI	5-10	-	Clear PCI request
	††001406	ECI	5-10	-	Enable PCI request
	††001407	DCI	5-10	-	Disable PCI request
	00200*k*	VL A*k*	5-12	-	Transmit (A*k*) to VL register
	†002000	VL 1	5-12	-	Transmit 1 to VL register
	002100	EFI	5-13	-	Enable interrupt on floating-point error
	002200	DFI	5-13	-	Disable interrupt on floating-point error
★	002300	ERI	5-13	-	Enable operand range interrupts

† Special syntax form
†† Privileged to monitor mode
★ New instructions
→ Deletions from Cray-1/S CAL

	CRAY-1	CAL	PAGE	UNIT	DESCRIPTION
★	002400	DRI	5-13	-	Disable operand range interrupts
★	002500	DBM	5-13	-	Disable bidirectional memory transfers
★	002600	EBM	5-13	-	Enable bidirectional memory transfers
★	002700	CMR	5-13	-	Complete memory references
	0030*j*0	VM Sj	5-15	-	Transmit (S*j*) to VM register
	†003000	VM 0	5-15	-	Clear VM register
★	0034*jk*	SM*jk* 1,TS	5-15	-	Test & set semaphore *jk* in SM
★	0036*jk*	SM*jk* 0	5-15	-	Clear semaphore *jk* in SM
★	0037*jk*	SM*jk* 1	5-15	-	Set semaphore *jk* in SM
→	004000	EX	5-17	-	Normal exit
	0050*jk*	J B*jk*	5-18	-	Jump to (B*jk*)
	006*ijkm*	J *exp*	5-19	-	Jump to *exp*
	007*ijkm*	R *exp*	5-20	-	Return jump to *exp*; set B00 to P.
	010*ijkm*	JAZ *exp*	5-21	-	Branch to *exp* if (A0)=0
	011*ijkm*	JAN *exp*	5-21	-	Branch to *exp* if (A0)≠0
	012*ijkm*	JAP *exp*	5-21	-	Branch to *exp* if (A0) positive; 0 is positive.
	013*ijkm*	JAM *exp*	5-21	-	Branch to *exp* if (A0) negative
	014*ijkm*	JSZ *exp*	5-23	-	Branch to *exp* if (S0)=0
	015*ijkm*	JSN *exp*	5-23	-	Branch to *exp* if (S0)≠0
	016*ijkm*	JSP *exp*	5-23	-	Branch to *exp* if (S0) positive; 0 is positive.
	017*ijkm*	JSM *exp*	5-23	-	Branch to *exp* if (S0) negative
	020*ijkm*	A*i* *exp*	5-25	-	Transmit *exp*=*jkm* to A*i*
	021*ijkm*	A*i* *exp*	5-25	-	Transmit *exp*=ones complement of *jkm* to A*i*
	022*ijk*	A*i* *exp*	5-26	-	Transmit *exp*=*jk* to A*i*
	023*ij*0	A*i* S*j*	5-27	-	Transmit (S*j*) to A*i*
★	023*i*01	A*i* VL	5-27	-	Transmit (VL) to A*i*
	024*ijk*	A*i* B*jk*	5-28	-	Transmit (B*jk*) to A*i*
	025*ijk*	B*jk* A*i*	5-28	-	Transmit (A*i*) to B*jk*
	026*ij*0	A*i* PS*j*	5-29	Pop/LZ	Population count of (S*j*) to A*i*
	026*ij*1	A*i* QS*j*	5-29	Pop/LZ	Population count parity of (S*j*) to A*i*
★	026*ij*7	A*i* SB*j*	5-29	-	Transmit (SB*j*) to A*i*
	027*ij*0	A*i* ZS*j*	5-31	Pop/LZ	Leading zero count of (S*j*) to A*i*

† Special syntax form

CRAY-1	CAL	PAGE	UNIT	DESCRIPTION
027*ij*7	SB*j* A*i*	5-31	-	Transmit (A*i*) to SB*j*
030*ijk*	A*i* A*j*+A*k*	5-32	A Int Add	Integer sum of (A*j*) and (A*k*) to A*i*
†030*i*0*k*	A*i* A*k*	5-32	A Int Add	Transmit (A*k*) to A*i*
†030*ij*0	A*i* A*j*+1	5-32	A Int Add	Integer sum of (A*j*) and 1 to A*i*
031*ijk*	A*i* A*j*-A*k*	5-32	A Int Add	Integer difference of (A*j*) less (A*k*) to A*i*
†031*i*00	A*i* -1	5-32	A Int Add	Transmit -1 to A*i*
†031*i*0*k*	A*i* -A*k*	5-32	A Int Add	Transmit the negative of (A*k*) to A*i*
†031*ij*0	A*i* A*j*-1	5-32	A Int Add	Integer difference of (A*j*) less 1 to A*i*
032*ijk*	A*i* A*j**A*k*	5-33	A Int Mult	Integer product of (A*j*) and (A*k*) to A*i*
033*i*00	A*i* CI	5-34	-	Channel number to A*i* (*j*=0)
033*ij*0	A*i* CA,A*j*	5-34	-	Address of channel (A*j*) to A*i* (*j*≠0; *k*=0)
033*ij*1	A*i* CE,A*j*	5-34	-	Error flag of channel (A*j*) to A*i* (*j*≠0; *k*=1)
034*ijk*	B*jk*,A*i* ,A0	5-36	Memory	Read (A*i*) words to B register *jk* from (A0)
†034*ijk*	B*jk*,A*i* 0,A0	5-36	Memory	Read (A*i*) words to B register *jk* from (A0)
035*ijk*	,A0 B*jk*,A*i*	5-36	Memory	Store (A*i*) words at B register *jk* to (A0)
†035*ijk*	0,A0 B*jk*,A*i*	5-36	Memory	Store (A*i*) words at B register *jk* to (A0)
036*ijk*	T*jk*,A*i* ,A0	5-36	Memory	Read (A*i*) words to T register *jk* from (A0)
†036*ijk*	T*jk*,A*i* 0,A0	5-36	Memory	Read (A*i*) words to T register *jk* from (A0)
037*ijk*	,A0 T*jk*,A*i*	5-36	Memory	Store (A*i*) words at T register *jk* to (A0)
†037*ijk*	0,A0 T*jk*,A*i*	5-36	Memory	Store (A*i*) words at T register *jk* to (A0)
040*ijkm*	S*i* *exp*	5-39	-	Transmit *jkm* to S*i*
041*ijkm*	S*i* *exp*	5-39	-	Transmit *exp*=ones complement of *jkm* to S*i*
042*ijk*	S*i* <*exp*	5-40	S Logical	Form ones mask *exp* bits in S*i* from the right; *jk* field gets 64-*exp*.
†042*ijk*	S*i* #>*exp*	5-40	S Logical	Form zeros mask *exp* bits in S*i* from the left; *jk* field gets 64-*exp*.
†042*i*77	S*i* 1	5-40	S Logical	Enter 1 into S*i*

† Special syntax form

✓ Functional Unit time changed from Cray-1/S

CRAY-1	CAL	PAGE	UNIT	DESCRIPTION
†042*i*00	S*i* -1	5-40	S Logical	Enter -1 into S*i*
043*ijk*	S*i* >*exp*	5-40	S Logical	Form ones mask *exp* bits in S*i* from the left; *jk* field gets *exp*.
†043*ijk*	S*i* #<*exp*	5-40	S Logical	Form zeros mask *exp* bits in S*i* from the right; *jk* field gets 64-*exp*.
†043*i*00	S*i* 0	5-40	S Logical	Clear S*i*
044*ijk*	S*i* S*j*&S*k*	5-41	S Logical	Logical product of (S*j*) and (S*k*) to S*i*
†044*ij*0	S*i* S*j*&SB	5-41	S Logical	Sign bit of (S*j*) to S*i*
†044*ij*0	S*i* SB&S*j*	5-41	S Logical	Sign bit of (S*j*) to S*i* (*j*≠0)
045*ijk*	S*i* #S*k*&S*j*	5-41	S Logical	Logical product of (S*j*) and ones complement of (S*k*) to S*i*
†045*ij*0	S*i* #SB&S*j*	5-41	S Logical	(S*j*) with sign bit cleared to S*i*
046*ijk*	S*i* S*j*\S*k*	5-41	S Logical	Logical difference of (S*j*) and (S*k*) to S*i*
†046*ij*0	S*i* S*j*\SB	5-41	S Logical	Toggle sign bit of S*j*, then enter into S*i*
†046*ij*0	S*i* SB\S*j*	5-41	S Logical	Toggle sign bit of S*j*, then enter into S*i* (*j*≠0)
047*ijk*	S*i* #S*j*\S*k*	5-41	S Logical	Logical equivalence of (S*k*) and (S*j*) to S*i*
†047*i*0*k*	S*i* #S*k*	5-41	S Logical	Transmit ones complement of (S*k*) to S*i*
†047*ij*0	S*i* #S*j*\SB	5-41	S Logical	Logical equivalence of (S*j*) and sign bit to S*i*
†047*ij*0	S*i* #SB\S*j*	5-41	S Logical	Logical equivalence of (S*j*) and sign bit to S*i* (*j*≠0)
†047*i*00	S*i* #SB	5-41	S Logical	Enter ones complement of sign bit into S*i*
050*ijk*	S*i* S*j*!S*i*&S*k*	5-41	S Logical	Logical product of (S*i*) and (S*k*) complement ORed with logical product of (S*j*) and (S*k*) to S*i*
†050*ij*0	S*i* S*j*!S*i*&SB	5-41	S Logical	Scalar merge of (S*i*) and sign bit of (S*j*) to S*i*
051*ijk*	S*i* S*j*!S*k*	5-41	S Logical	Logical sum of (S*j*) and (S*k*) to S*i*
†051*i*0*k*	S*i* S*k*	5-41	S Logical	Transmit (S*k*) to S*i*
†051*ij*0	S*i* S*j*!SB	5-41	S Logical	Logical sum of (S*j*) and sign bit to S*i*

† Special syntax form

CRAY-1	CAL	PAGE	UNIT	DESCRIPTION
†051*ij*0	*Si* SB!*Sj*	5-41	S Logical	Logical sum of (S*j*) and sign bit to S*i* (*j*≠0)
†051*i*00	*Si* SB	5-41	S Logical	Enter sign bit into S*i*
052*ijk*	S0 *Si<exp*	5-45	S Shift	Shift (S*i*) left *exp*=*jk* places to S0
053*ijk*	S0 *Si>exp*	5-45	S Shift	Shift (S*i*) right *exp*=64-*jk* places to S0
054*ijk*	*Si Si<exp*	5-45	S Shift	Shift (S*i*) left *exp*=*jk* places
055*ijk*	*Si Si>exp*	5-45	S Shift	Shift (S*i*) right *exp*=64-*jk* places
056*ijk*	*Si Si,Sj<Ak*	5-46	S Shift	Shift (S*i* and S*j*) left (A*k*) places to S*i*
†056*ij*0	*Si Si,Sj<1*	5-46	S Shift	Shift (S*i* and S*j*) left one place to S*i*
†056*i*0*k*	*Si Si<Ak*	5-46	S Shift	Shift (S*i*) left (A*k*) places to S*i*
057*ijk*	*Si Sj,Si>Ak*	5-46	S Shift	Shift (S*j* and S*i*) right (A*k*) places to S*i*
†057*ij*0	*Si Sj,Si>1*	5-46	S Shift	Shift (S*j* and S*i*) right one place to S*i*
†057*i*0*k*	*Si Si>Ak*	5-46	S Shift	Shift (S*i*) right (A*k*) places to S*i*
060*ijk*	*Si* *Sj*+*Sk*	5-48	S Int Add	Integer sum of (S*j*) and (S*k*) to S*i*
061*ijk*	*Si* *Sj*-*Sk*	5-48	S Int Add	Integer difference of (S*j*) and (S*k*) to S*i*
†061*i*0*k*	*Si* -*Sk*	5-48	S Int Add	Transmit negative of (S*k*) to S*i*
062*ijk*	*Si* *Sj*+FS*k*	5-49	Fp Add	Floating-point sum of (S*j*) and (S*k*) to S*i*
†062*i*0*k*	*Si* +FS*k*	5-49	Fp Add	Normalize (S*k*) to S*i*
063*ijk*	*Si* *Sj*-FS*k*	5-49	Fp Add	Floating-point difference of (S*j*) and (S*k*) to S*i*
†063*i*0*k*	*Si* -FS*k*	5-49	Fp Add	Transmit normalized negative of (S*k*) to S*i*
064*ijk*	*Si* *Sj**FS*k*	5-51	Fp Mult	Floating-point product of (S*j*) and (S*k*) to S*i*
065*ijk*	*Si* *Sj**HS*k*	5-51	Fp Mult	Half-precision rounded floating-point product of (S*j*) and (S*k*) to S*i*
066*ijk*	*Si* *Sj**RS*k*	5-51	Fp Mult	Full-precision rounded floating-point product of (S*j*) and (S*k*) to S*i*

† Special syntax form

CRAY-1	CAL	PAGE	UNIT	DESCRIPTION
067*ijk*	S*i* S*j**IS*k*	5-51	Fp Mult	2-floating-point product of (S*j*) and (S*k*) to S*i*
070*ij*0	S*i* /HS*j*	5-53	Fp Rcpl	Floating-point reciprocal approximation of (S*j*) to S*i*
071*i*0*k*	S*i* A*k*	5-54	-	Transmit (A*k*) to S*i* with no sign extension
071*i*1*k*	S*i* +A*k*	5-54	-	Transmit (A*k*) to S*i* with sign extension
071*i*2*k*	S*i* +FA*k*	5-54	-	Transmit (A*k*) to S*i* as unnormalized floating-point number
071*i*30	S*i* 0.6	5-54	-	Transmit constant 0.75*2**48 to S*i*
071*i*40	S*i* 0.4	5-54	-	Transmit constant 0.5 to S*i*
071*i*50	S*i* 1.	5-54	-	Transmit constant 1.0 to S*i*
071*i*60	S*i* 2.	5-54	-	Transmit constant 2.0 to S*i*
071*i*70	S*i* 4.	5-54	-	Transmit constant 4.0 to S*i*
072*i*00	S*i* RT	5-56	-	Transmit (RTC) to S*i*
★ 072*i*02	S*i* SM	5-56	-	Transmit (SM) to S*i*
★ 072*ij*3	S*i* ST*j*	5-56	-	Transmit (ST*j*) to S*i*
073*i*00	S*i* VM	5-56	-	Transmit (VM) to S*i*
★ 073*ij*1	S*i* SR*j*	5-56	-	Transmit (SR*j*) to S*i* (*j*=0)
★ 073*i*02	SM S*i*	5-56	-	Transmit (S*i*) to SM
★ 073*ij*3	ST*j* S*i*	5-56	-	Transmit (S*i*) to ST*j*
074*ijk*	S*i* T*jk*	5-56	-	Transmit (T*jk*) to S*i*
075*ijk*	T*jk* S*i*	5-56	-	Transmit (S*i*) to T*jk*
076*ijk*	S*i* V*j*,A*k*	5-58	-	Transmit (V*j*, element (A*k*)) to S*i*
077*ijk*	V*i*,A*k* S*j*	5-58	-	Transmit (S*j*) to V*i* element (A*k*)
†077*i*0*k*	V*i*,A*k* 0	5-58	-	Clear V*i* element (A*k*)
10*hijkm*	A*i* *exp*,A*h*	5-59	Memory	Read from ((A*h*)+*exp*) to A*i* (A0=0)
†100*ijkm*	A*i* *exp*,0	5-59	Memory	Read from (*exp*) to A*i*
†100*ijkm*	A*i* *exp*,	5-59	Memory	Read from (*exp*) to A*i*
†10*hi*00 0	A*i* ,A*h*	5-59	Memory	Read from (A*h*) to A*i*
11*hijkm*	*exp*,A*h* A*i*	5-59	Memory	Store (A*i*) to (A*h*)+*exp* (A0=0)
†110*ijkm*	*exp*,0 A*i*	5-59	Memory	Store (A*i*) to *exp*
†110*ijkm*	*exp*, A*i*	5-59	Memory	Store (A*i*) to *exp*
†11*hi*00 0	,A*h* A*i*	5-59	Memory	Store (A*i*) to (A*h*)
12*hijkm*	S*i* *exp*,A*h*	5-59	Memory	Read from ((A*h*)+*exp*) to S*i* (A0=0)

† Special syntax form

CRAY-1	CAL	PAGE	UNIT	DESCRIPTION
†120*ijkm*	Si *exp*,0	5-59	Memory	Read from *exp* to Si
†120*ijkm*	Si *exp*,	5-59	Memory	Read from *exp* to Si
†12*hi*00 0	Si ,Ah	5-59	Memory	Read from (Ah) to Si
13*hijkm*	*exp*,Ah Si	5-59	Memory	Store (Si) to (Ah)+*exp* (A0=0)
†130*ijkm*	*exp*,0 Si	5-59	Memory	Store (Si) to *exp*
†130*ijkm*	*exp*, Si	5-59	Memory	Store (Si) to *exp*
†13*hi*00 0	,Ah Si	5-59	Memory	Store (Si) to (Ah)
140*ijk*	Vi Sj&Vk	5-61	V Logical	Logical products of (Sj) and (Vk) to Vi
141*ijk*	Vi Vj&Vk	5-61	V Logical	Logical products of (Vj) and (Vk) to Vi
142*ijk*	Vi Sj!Vk	5-61	V Logical	Logical sums of (Sj) and (Vk) to Vi
†142*i*0*k*	Vi Vk	5-61	V Logical	Transmit (Vk) to Vi
143*ijk*	Vi Vj!Vk	5-61	V Logical	Logical sums of (Vj) and (Vk) to Vi
144*ijk*	Vi Sj\Vk	5-61	V Logical	Logical differences of (Sj) and (Vk) to Vi
145*ijk*	Vi Vj\Vk	5-61	V Logical	Logical differences of (Vj) and (Vk) to Vi
†145*iii*	Vi 0	5-61	V Logical	Clear Vi
146*ijk*	Vi Sj!Vk&VM	5-61	V Logical	Transmit (Sj) if VM bit=1; (Vk) if VM bit=0 to Vi.
†146*i*0*k*	Vi #VM&Vk	5-61	V Logical	Vector merge of (Vk) and 0 to Vi
147*ijk*	Vi Vj!Vk&VM	5-61	V Logical	Transmit (Vj) if VM bit=1; (Vk) if VM bit=0 to Vi.
150*ijk*	Vi Vj<Ak	5-65	V Shift	Shift (Vj) left (Ak) places to Vi
†150*ij*0	Vi Vj<1	5-65	V Shift	Shift (Vj) left one place to Vi
151*ijk*	Vi Vj>Ak	5-65	V Shift	Shift (Vj) right (Ak) places to Vi
†151*ij*0	Vi Vj>1	5-65	V Shift	Shift (Vj) right one place to Vi
152*ijk*	Vi Vj,Vj<Ak	5-67	V Shift	Double shift (Vj) left (Ak) places to Vi
†152*ij*0	Vi Vj,Vj<1	5-67	V Shift	Double shift (Vj) left one place to Vi
153*ijk*	Vi Vj,Vj>Ak	5-67	V Shift	Double shift (Vj) right (Ak) places to Vi
†153*ij*0	Vi Vj,Vj>1	5-67	V Shift	Double Shift (Vj) right one place to Vi

† Special syntax form

CRAY-1	CAL	PAGE	UNIT	DESCRIPTION
154*ijk*	V*i* S*j*+V*k*	5-72	V Int Add	Integer sums of (S*j*) and (V*k*) to V*i*
155*ijk*	V*i* V*j*+V*k*	5-72	V Int Add	Integer sums of (V*j*) and (V*k*) to V*i*
156*ijk*	V*i* S*j*-V*k*	5-72	V Int Add	Integer differences of (S*j*) and (V*k*) to V*i*
†156*i*0*k*	V*i* -V*k*	5-72	V Int Add	Transmit negative of (V*k*) to V*i*
157*ijk*	V*i* V*j*-V*k*	5-72	V Int Add	Integer differences of (V*j*) and (V*k*) to V*i*
160*ijk*	V*i* S*j**FV*k*	5-74	Fp Mult	Floating-point products of (S*j*) and (V*k*) to V*i*
161*ijk*	V*i* V*j**FV*k*	5-74	Fp Mult	Floating-point products of (V*j*) and (V*k*) to V*i*
162*ijk*	V*i* S*j**HV*k*	5-74	Fp Mult	Half-precision rounded floating-point products of (S*j*) and (V*k*) to V*i*
163*ijk*	V*i* V*j**HV*k*	5-74	Fp Mult	Half-precision rounded floating-point products of (V*j*) and (V*k*) to V*i*
164*ijk*	V*i* S*j**RV*k*	5-74	Fp Mult	Rounded floating-point products of (S*j*) and (V*k*) to V*i*
165*ijk*	V*i* V*j**RV*k*	5-74	Fp Mult	Rounded floating-point products of (V*j*) and (V*k*) to V*i*
166*ijk*	V*i* S*j**IV*k*	5-74	Fp Mult	2-floating-point products of (S*j*) and (V*k*) to V*i*
167*ijk*	V*i* V*j**IV*k*	5-74	Fp Mult	2-floating-point products of (V*j*) and (V*k*) to V*i*
170*ijk*	V*i* S*j*+FV*k*	5-77	Fp Add	Floating-point sums of (S*j*) and (V*k*) to V*i*
†170*i*0*k*	V*i* +FV*k*	5-77	Fp Add	Normalize (V*k*) to V*i*
171*ijk*	V*i* V*j*+FV*k*	5-77	Fp Add	Floating-point sums of (V*j*) and (V*k*) to V*i*
172*ijk*	V*i* S*j*-FV*k*	5-77	Fp Add	Floating-point differences of (S*j*) and (V*k*) to V*i*
†172*i*0*k*	V*i* -FV*k*	5-77	Fp Add	Transmit normalized negatives of (V*k*) to V*i*
173*ijk*	V*i* V*j*-FV*k*	5-77	Fp Add	Floating-point differences of (V*j*) and (V*k*) to V*i*
174*ij*0	V*i* /HV*j*	5-79	Fp Rcpl	Floating-point reciprocal approximations of (V*j*) to V*i*

† Special syntax form

CRAY-1	CAL	PAGE	UNIT	DESCRIPTION
174*ij*1	V*i* PV*j*	5-80	V Pop	Population counts of (V*j*) to V*i*
174*ij*2	V*i* QV*j*	5-80	V Pop	Population count parities of (V*j*) to V*i*
1750*j*0	VM V*j*,Z	5-82	V Logical	VM=1 where (V*j*)=0
1750*j*1	VM V*j*,N	5-82	V Logical	VM=1 where (V*j*)≠0
1750*j*2	VM V*j*,P	5-82	V Logical	VM=1 if (V*j*) positive; 0 is positive.
1750*j*3	VM V*j*,M	5-82	V Logical	VM=1 if (V*j*) negative
176*i*0*k*	V*i* ,A0,A*k*	5-84	Memory	Read (VL) words to V*i* from (A0) incremented by (A*k*)
†176*i*00	V*i* ,A0,1	5-84	Memory	Read (VL) words to V*i* from (A0) incremented by 1
1770*jk*	,A0,A*k* V*j*	5-84	Memory	Store (VL) words from V*j* to (A0) incremented by (A*k*)
†1770*j*0	,A0,1 V*j*	5-84	Memory	Store (VL) words from V*j* to (A0) incremented by 1

† Special syntax form

APPENDIX C

GLOSSARY

4/7/83

X-MP GLOSSARY

Activation Record (AR) - The element of a TASKSTACK associated with a subroutine call from within the task. An activation block/record contains: traceback addresses and local variable storage locations.

Asynchronous - A mode of operation in which performance of operations is not dependent on the completion of all previous operations. Asynchronous I/O using Buffer In and Buffer Out instructions allows process to continue without wating for I/O to complete. (The use of Unit and Length functions set synchronization points thus changing the mode of operation back to synchronous.) The firing up (starting) of a task makes the mode of operation of a program or job asynchronous since processing will continue without waiting for the completion of the task. (The use of Events, Locks and TSKWAIT subroutines set synchronization points and may thus change mode of operation of a multitask process back to synchronous.)

Blank Common Block - A common block into which data cannot be initialized at load time. Any number of program modules may declare a blank common block and each may declare a block of a different size. The loader allocates storage to the blank common block after all other blocks have been processed.

Chime - A series of pipelined instructions. The execution time for the chime is dominated by the execution time of first instruction in the sequence. Overlapping of execution times of subsequent instructions in the chime diminishes their cost.

Common Block - A block of data that can be declared by more than one program module. More than one program module can specify initial values for data in a common block but if a conflict occurs, information from later programs is loaded over previously loaded information. The two types of common blocks are labeled and blank.

Critical Region - A part of a sequential program operating on shared data in such a way that it must have exclusive access to the shared data during its execution.

Deadlock - A state resulting in the inability of processing to continue due to an unresolvable conflict. Waiting for something to happen that cannot happen results in a deadlock.

Deadly Embrace - A special case of a deadlock involving two interactive processes. Process A is waiting for process B to do something and process B is waiting for process A to do something; neither can break its own wait cycle.

Deterministic - A property of a process which allows any future state of the process to be predicted exactly. Repeated executions of a deterministic software process will always produce the same results in the same amount of time.

Event - A signal indicating an action of significance to other tasks of the same job. One means of coordinating multiple tasks is through the signaling of (posting) and waiting for (testing) an event. (*EVENTMARK)

Exchange - A mechanism for facilitating the contact switch between tasks (i.e., software processes).

Exchange Package - A 16-word block of data in an area of memory that is reserved for exchange packages. This block of data contains the contents of all of the necessary registers and conditions or mode flags which are associated with a particular program. Each program residing in memory will have an associated exchange package (refer to the CRAY-1 Hardware Reference Manual).

Execution Point - The instruction of the code associated with a task that is pointed to by the P register in an exchange package. Every task has an execution point.

Fork and Join - Transition points where the nature of a process changes from serial (sequential) to parallel (FORK) and from parallel to serial (JOIN).

Global Variable - A variable whose value is accessible throughout a program.

Instruction Stream - A series of instructions to be pointed to and executed sequentially as a block of code. An instruction stream may be defined to be a task if it can be executed in parallel with another instruction stream. Instruction streams do not have their own exchange package but tasks do.

Job - (1) An arbitrarily defined parcel of work submitted to a computing system. (2) A collection of tasks submitted to the system and treated by the system as an entity. With respect to a job, the system is parametrically controlled by the content of the job dataset.

Job Step - A unit of work within a job, such as source language compilation or object program execution. Instructions to be executed and data associated with a particular control statement are parts of a job step.

Local Variable - A variable whose value is known only to the program module in which it is defined. It exists only during the execution of that module and may not be accessed or modified by other modules.

Lock - A mechanism for assuring that unique access to data can be secured. If a process encounters a lock on memory it wishes access it must wait for it to be unlocked before it can continue.

Logical CPU - A scheduling unit. In COS this is associated with an exchange package. (*VIRTUAL PROCESSOR)

Loosely Coupled - A lower level of synchronization and communication required by software processes in a multitasking or multiprocessing environment.

Multiprocessing - The utilization of more than one processor to logically or functionally divide processes and to execute various segments in parallel Multiprocessing may be simulated by one processor in a multiprogramming environment.

Multiprogramming - A technique for handling numerous routines or programs simultaneously by interleaving their execution: i.e., permitting more than one program to share machine resources (COS 1.11 is a multiprogramming operating system using jobs as the unit of user work).

Multitasking - A technique in which several separate but interrelated tasks operate under a single program or job identity. It may or may not be a form of multiprocessing.

Overlaying - A technique for bringing routines into memory from some other form of storage during processing so that several routines will occupy the same storage locations at different times. Overlaying is used when the total memory requirement for instructions exceeds the available memory.

Parallel - Objects (tasks, job steps, elements of an array) considered simultaneously (or nearly so) rather than in sequence or in some special order.

Physical CPU - A processor (a real hardware entity).

Pipelining - The beginning of an operation before a previous one has been completed. Pipelining is accomplished on the Cray through the use of fully segmented functional units that allow several sets of operands to be at various stages of processing in the same functional unit at the same time.

Posting - The entering of a unit of information in a location to be examined for messages. An event is said to be posted when it is signaling some occurence having taken place. Event posting is done through a call to a library routine in the Cray system.

Ready - A state of a task in which it has fulfilled all conditions for its execution and is queued for scheduling of a logical CPU (*PENDING)

Reentrant - The property of a program module that allows one copy of it to be used by more than one job or task. A mechanism is supplied by which the routines environment is preserved, i.e., working storage and control indicators are assigned independent storage location each time the routine is called. Only reentrant code can recursive call itself.

Scheduling Unit - An entity that can be scheduled as an independent unit by a multiprogramming operating system (eg., tasks, jobs).

Scope of a Variable - That portion of code for which the variable is defined and in which it can be referenced. In FORTRAN the portion of code is the program, subprogram or statement.

Serially Reusable - The property of an instruction stream that allows one copy of it to be used by more than one job or task but only one at a time. The second task wishing to enter a serially reentrant code must wait if another user has entered first and not yet exited. The routines environment must be restored to its initial condition after each use. This is referred to as single threading of the code.

Shared Data - Data which may be referenced and modified by the program modules that share it.

Single Threading - Supporting only one user at a time. See Serially Reusable.

STACK - A data structure providing a dynamic sequential data list having special provisions for access from one end or the other. A last in, first out (push down, pop up) stack is accessed from just one end.

Starvation - A state of deprivation of a task in which it never gets a chance to execute.

Suspended - A state of a task in which it cannot be executed (i.e., it doesn't have possession of a logical CPU).

Synchronous - A mode of operation in which the performance of an operation does not begin until all previous operations are complete. The normal execution of FORTRAN code including I/O statements is synchronous. Calls to subroutine and function references in FORTRAN could be viewed as synchronous operations. Synchronous I/O hardware channels operate under the restriction that the ready (for output) or the resume (for input) signal is held on during data transfer.

Task - A subjob or subprogram. A unique process that may have code and data areas in common with other tasks of the same job. A task is treated as a scheduling unit in a multitasking environment.

Task Control Block - The area in user assigned memory, but not accessable to the user job, containing all the information associated with an active task (one that has to be started but has not yet encountered the stop or return). The contents include: the tasks exchange package, pointers to the TASKSTACK and subroutines containing task code.

TASKSTACK - A push down, pop up stack created upon the activation (firing up) of a task. The elements of the TASKSTACK are activation record. One activiation block is created (placed on top) each time a subroutine is called and popped off when STOP or RETURN is executed.

Temporary - Short term; for immediate use only; not made permanent by saving it for long term future retrieval.

Tightly Coupled - A higher degree of synchronization and communication (binding) required by software processes in a multitasking environment. Tasks may handle their own communication with other tasks of the same job.

*Term defined in the "Industrial Real-Time FORTRAN" standard
(IPW/EWICS TC1, 2.2/80)

APPENDIX D

QUESTIONS AND ANSWERS ABOUT MULTIPROCESSING

BY

DAVID E. WHITNEY

1. This paper is not a good user document.

   Agreed. It was culled from notes and working papers and is not intended to be released to users. User documents are being prepared, although they will not be ready for several more months.

2. These facilities are very primitive.

   This is intentional. It is expected that these facilities will be used to construct the building blocks that will be needed by an application program. By providing the primitives, a facility, such as a message queue, may be tailored to the characteristics of the application.

   It is expected that additional facilities will be provided as we gain experience with multitasking. It is our intention to provide facilities which are both useful and easy to use. Comments and suggestions will continue to be welcomed.

3. Will these facilities be available on single processer CRAY-1 systems?

   Yes. Without hardware support for conflict resolution, the underlying implementation may differ significantly from that on the X-MP. It is intended that source codes be portable between machines.

   We are currently viewing the ability to multitask on a single cpu machine as a development tool for customers to use while waiting for delivery of an X-MP.

4. The naming of the routines is inconsistent.

   True. The names of the locking routines have been changed to achieve some measure of consistency.

5. What does it cost to start a task?

   A lot. Starting a task can be the most expensive operation performed by a multitasking program. We recommend that this operation be minimized.

   The cost is due to two operations which can have unpredictable consequences: creating exchange packets and allocating task local memory space. For every cpu which may be utilized by the program there must be space allocated for an exchange packet, a register save area and various accounting and control structures required by the operating system. In addition to the operating system overhead needed to create and initialize these structures, it may be necessary to expand the memory size of the program.

The job of allocating task local memory may also require that the memory size of the program be expanded. At best, this will require memory to memory data transfers and at the worst, will require that the program be rolled to disk and rescheduled. Although a program may not be charged directly for this system overhead, there may be a noticable impact on wall clock time for the program and, possibly, on overall system throughput.

Two strategies are being investigated in an attempt to deal with this problem: pre-allocation and retention. Pre-allocation involves initializing a small number of unused exchange areas and reserving enough unused space so that a small number of task local stacks can be rapidly allocated. Retention involves hanging on to an exchange area or stack after a task terminates so that they can be used by a new task.

Retention is a strategy that can also be employed by a user's program. This could involve creating a task which acts as a service routine and does not terminate until its services are no longer required. Such a service task would wait on an event or a lock until another task wants to use its services. After the event is signaled or the lock is released, the service task can perform its function and then return to the wait state until it is brought into service again.

6. Is the main program a task?

Yes. It is special only because the loader creates it. It can do any of the operations permitted by a task. It may even terminate and disappear without effecting the continued execution of other tasks it may have created. The only restriction is that other tasks can not wait on the completion of the main program, nor may they inquire into the status of the main program since they do not have access to the programs unique task identifier.

7. How does a task terminate?

By executing any of the FORTRAN statements which would normally cause control to be passed outside the scope of a task: END, RETURN, STOP, CALL EXIT, CALL ABORT, or CALL ERREXIT.

An abnormal termination of a task will cause all other tasks to terminate as well.

8. Can a program recover from an abnormal condition?

Reprieve processing can be used. No additional facilities are provided by the multitasking support library.

9. Can relative priorities be attached to tasks?

No. Relative priorities is an inadequate mechanism for controlling

tasks in an order dependent algorithm. Locks or events must be used to guarantee proper sequencing of execution.

10. How do tasks communicate?

    Through shared global memory. No special mechanism is provided, although message queues can be constructed from the lock and event primitives which have been provided.

11. Can more information about a task's state be returned from TSKTEST?

    No. The reality of external interrupts demands a preference for asynchronously executing tasks. Knowing that a task is "currently executing", "ready to execute", or "waiting on something" leads to unnecessarily complex programs which use unreliable information to make decisions.

12. What does it cost to use a lock?

    The cost is composed of the call overhead and several scalar memory references.

13. A special, fast lock mechanism would be advantageous.

    It might be, if it were safe. The proposal to allow a set of predefined locks has been dropped because it encourages an undisciplined use of locks which may cause the problems locks are intended to prevent.

    A second reason for dropping this proposal is that it would not be any faster, unless the compiler generated inline code for it. We are not yet prepared to do that.

14. How will multi-state semaphores be implemented?

    By the programmer. They are a user defined data structure composed of a lock and an integer variable. The multitasking library knows nothing about this Dijkstra type of semaphore. The use of a data structure to simulate them is done by program conventions that can use the lock mechanism.

15. The LOCKTEST function should set the lock before returning.

    The definition of the function has been changed to do this, since it seems to make the function more useable.

16. Is it possible to determine that a lock does not exist?

No. The modifications made to the lock mechanisms treat locks as data variables. As long as storage exists, the lock exists; even when the variable contains an invalid or meaningless value. It is important that a lock be initialized with a call to LOCKASGN before any other references to it.

17. What does it cost to use events?

The cost of using an event is comparable to the cost of using a lock except for one important case: waiting on an event that has not been posted. This case requires additional overhead to switch the cpu to another task and to queue the waiting task on the event. This may require a call to the operating system and a trip through the scheduler.

18. What happens when an event is posted?

All the tasks waiting on the event are moved to the scheduling queue to compete for cpu access. In addition, a value is placed in the event identifier so that any other requests to wait on that event will be satisfied immediately when they are made, eliminating any need to queue the requesting task.

The cost of activating a waiting task may involve a call to the operating system and a trip thrcugh the scheduler.

19. Aren't events cleared and reset after an EVWAIT is satisfied?

No. Any number of EVWAITS may be satisfied by a single EVPOST. This is an important difference between locks and events. Certain uses of events may even require that the event be protected with a lock in order to assure correct execution of the program.

20. Is it possible to determine that an event does not exist?

No. As with locks, events are data variables that behave in a particular way. As long as storage exists, the event exists; even when the variable contains an invalid or meaningless value. It is important that an event be initialized with a call to EVASGN before any other references to it.

21. Why are both locks and events included?

An easy to use facility needs both a sequential access mechanism and a broadcast mechanism for co-ordinating tasks. By biasing the implementations, programs can also indicate a short or long

wait. It is possible to use locks to simulate events and events to simulate locks with appropriate, although awkward, programming conventions.

We expect that most locks will be used to protect access to shared data and that most events will be used to indicate that a particular state of processing that data has been reached. With experience, we may be able to refine the distinction between locks and events.

22. What is the purpose of LOCKASGN and EVASGN?

To set unique initial values in a lock or event variable.

23. What is the purpose of LOCKREL and EVREL?

To indicate to the multitasking library that the lock or event variable will no longer be used. Any tasks waiting on the variable are in error. Any further attempt to use the lock or event in the multitasking library is an error.

24. What is the purpose of TSKTEST, LOCKTEST and EVTEST?

To allow a condition block to be executed. One possibility is suggested by the retention strategy mentioned in question 5. It is not meant to provide a vehicle for dynamically reconfigurable program structures.

25. Is there a limit placed on the number of tasks, locks or events which can be active?

No. Locks and events require no additional storage space beyond what is allocated for them by the compiler. Space for new tasks will be allocated as requested until there is no more memory available on the machine.

26. What alternative has been provided to expanding blank common?

A heap oriented storage management facility will be provided. Once a program has determined how much storage it needs, it will be allocated from the same pool of space that is used to allocate stack space for new tasks. When a program has finished using this space it can be returned to the pool and reused later.

Programmers will need to use the POINTER facility in CFT to bind dynamically allocated space to an array declaration. In addition, programmers can not allocate two different items with two separate calls to the storage manager and expect the allocated space to be contiguous.

27. How can I be sure that local variables are allocated in the task local data area?

This is tricky because there is no direct way to specify local allocation. The CFT manual should be consulted for the specific rules that govern storage allocation with a stack and the new calling sequence. Basically, items are local unless specified global with a COMMON or STATIC declaration.

28. What register conventions are followed by the multitasking library?

The standard format is followed: all A, S and V registers are assumed to be available; B and T registers are saved and restored, except for those dedicated as scratch registers. In addition, some of the hardware semaphore bits are reserved for use in the library, and it is expected that some of the shared B and T registers will be used in the future. To avoid future conflicts, programmers should avoid use of the cluster registers.

29. Have the rules changed for using the math library?

No. The math library is being made re-entrant and can be used by any number of tasks at the same time. Register conventions needed to interface to the routines will remain the same; although some internal changes in register usage may cause problems for CAL routines that "happen to know that a certain value is in A4" or that "S6 is never used" by a particular routine.

30. Does the programmer need to worry about the integrity of the I/O library?

No. The library is being modified to protect itself from multiple, simultaneous access by different tasks. This will protect the transfer of records from or to a file.

31. How are files shared among tasks?

Between records. Once the library accepts a Begin I/O request from a task, all other tasks are prevented from accessing the file until the original task issues an End I/O command. On the FORTRAN level, this means that each I/O statement is protected, but that there is no protection between statements. If the algorithms being programmed require that a group of records be read or written in a specific order, it will be necessary for the programmer to establish another level of access protection for a file, with the use of locks or events.

32. Can I control the size of a task's local data area?

wait. It is possible to use locks to simulate events and events to simulate locks with appropriate, although awkward, programming conventions.

We expect that most locks will be used to protect access to shared data and that most events will be used to indicate that a particular state of processing that data has been reached. With experience, we may be able to refine the distinction between locks and events.

22. What is the purpose of LOCKASGN and EVASGN?

To set unique initial values in a lock or event variable.

23. What is the purpose of LOCKREL and EVREL?

To indicate to the multitasking library that the lock or event variable will no longer be used. Any tasks waiting on the variable are in error. Any further attempt to use the lock or event in the multitasking library is an error.

24. What is the purpose of TSKTEST, LOCKTEST and EVTEST?

To allow a condition block to be executed. One possibility is suggested by the retention strategy mentioned in question 5. It is not meant to provide a vehicle for dynamically reconfigurable program structures.

25. Is there a limit placed on the number of tasks, locks or events which can be active?

No. Locks and events require no additional storage space beyond what is allocated for them by the compiler. Space for new tasks will be allocated as requested until there is no more memory available on the machine.

26. What alternative has been provided to expanding blank common?

A heap oriented storage management facility will be provided. Once a program has determined how much storage it needs, it will be allocated from the same pool of space that is used to allocate stack space for new tasks. When a program has finished using this space it can be returned to the pool and reused later.

Programmers will need to use the POINTER facility in CFT to bind dynamically allocated space to an array declaration. In addition, programmers can not allocate two different items with two separate calls to the storage manager and expect the allocated space to be contiguous.

27. How can I be sure that local variables are allocated in the task local data area?

This is tricky because there is no direct way to specify local allocation. The CFT manual should be consulted for the specific rules that govern storage allocation with a stack and the new calling sequence. Basically, items are local unless specified global with a COMMON or STATIC declaration.

28. What register conventions are followed by the multitasking library?

The standard format is followed: all A, S and V registers are assumed to be available; B and T registers are saved and restored, except for those dedicated as scratch registers. In addition, some of the hardware semaphore bits are reserved for use in the library, and it is expected that some of the shared B and T registers will be used in the future. To avoid future conflicts, programmers should avoid use of the cluster registers.

29. Have the rules changed for using the math library?

No. The math library is being made re-entrant and can be used by any number of tasks at the same time. Register conventions needed to interface to the routines will remain the same; although some internal changes in register usage may cause problems for CAL routines that "happen to know that a certain value is in A4" or that "S6 is never used" by a particular routine.

30. Does the programmer need to worry about the integrity of the I/O library?

No. The library is being modified to protect itself from multiple, simultaneous access by different tasks. This will protect the transfer of records from or to a file.

31. How are files shared among tasks?

Between records. Once the library accepts a Begin I/O request from a task, all other tasks are prevented from accessing the file until the original task issues an End I/O command. On the FORTRAN level, this means that each I/O statement is protected, but that there is no protection between statements. If the algorithms being programmed require that a group of records be read or written in a specific order, it will be necessary for the programmer to establish another level of access protection for a file, with the use of locks or events.

32. Can I control the size of a task's local data area?

Not from the task. Generation parameters in the library can be used to determine stack size; however, every task will be allocated the same amount of space. It may be possible to introduce an option on the TSKSTART call, at some future date, when we have gained more experience with this facility.

33. How are tasks scheduled for CPU utilization when there are other jobs in the system?

By priorities. The priority of a job is used for every task within the job. It is very possible for a task within a job to be given a CPU while all the other CPU's are in use by other, unrelated jobs.

34. Can a task be rolled out?

Not by itself. Roll out is done on jobs, so all activity, by every task within the job, must terminate before a job can be rolled.

35. How are deadlocks detected?

Two events are detected by the operating system which might imply a deadlock: a limit is reached on resource utilization or the job fails to use any resources. Both are done on a job basis. The first condition is the normal limiting scheme used on batch jobs. The second results when all tasks within a job have put themselves into a wait state either on a lock or an event. No attempts are made by the multitasking library to detect deadlocks.

READERS COMMENT FORM

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME _____

JOB TITLE _____

FIRM _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

CRAY RESEARCH, INC.

READERS COMMENT FORM

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME _____

JOB TITLE _____

FIRM _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

CRAY RESEARCH, INC.