TECHNICAL REPORT RD-GC-88-38

A DESCRIPTION OF THE COMPUTER SIMULATION OF A NEW BUS ARBITRATION SCHEME

G. Patton Bradford
Guidance & Control Directorate
Research, Development, & Engineering Center

DECEMBER 1988

U.S. ARMY MISSILE COMMAND
Redstone Arsenal, Alabama 35899-5000

Approved for public release; distribution unlimited.
**A DESCRIPTION OF THE COMPUTER SIMULATION OF A NEW BUS ARBITRATION SCHEME (U)**

**G. Patton Bradford**

**Date of Report: December 1988**

**Page Count:** 30

**Abstract:**

The object of bus arbitration schemes is to provide fair access to the bus by the various elements of the computer system. Arbitration must allow for each element to have access to the bus while, at the same time, preventing any single element from monopolizing the bus. The method of bus arbitration presented here appears to meet these goals.

The arbitration scheme described in this report places the various elements of the system on a rotating system of priorities. After the various priorities are set initially (0...N-1, where N is the number of system elements) all the priorities are incremented by one on a modulo N basis. In this way, each processor will eventually have top priority and be able to take possession of the bus if needed.

In a simulation of this scheme using a random mix of bus demands, the average wait for any processor for the bus was very short, generally less than 10 bus cycles ("bus cycle" defined in the body of the report.) For simulations in which the bus demand was high (a probability...)

**Subject Terms:**

Computer bus, computer architecture, arbitration

<table>
<thead>
<tr>
<th>Field</th>
<th>Group</th>
<th>Sub-Group</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Distribution/Availability of Abstract:**

UNCLASSIFIED

**Accession Number:** ANGNI-RD-GC-S

**Office Symbol:** ANGNI-RD-GC-S

**Security Classification:**

UNCLASSIFIED
Item 19. Abstract

of requesting the bus of 0.75 and a probability of retaining the bus of 0.75),
the average delay was seven cycles.

The simulation program itself is described in detail and the results of
various combinations of probabilities are presented.
ACKNOWLEDGEMENTS

I must be honest and state right from the beginning that this arbitration scheme is not mine; I did not think of it, and I can in no way lay claim to it. This scheme is the idea of Mr. James Holeman of Systems Dynamics, Inc. During some discussions on the design of an experimental digital signal processing board, Jim presented this idea as a simple, reliable, and robust method for arbitrating access to a common bus between various system boards. This simulation appears to bear out some of his hopes for the design in that it provides for short average delays as well as a relative immunity to system failure due to the failure of system components.

Thanks, Jim. I appreciate it.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II. ARBITRATION ALGORITHM DESCRIPTION</td>
<td>2</td>
</tr>
<tr>
<td>III. BUS SIMULATION PROGRAM</td>
<td>3</td>
</tr>
<tr>
<td>IV. SIMULATION RESULTS</td>
<td>4</td>
</tr>
<tr>
<td>V. CONCLUSIONS</td>
<td>5</td>
</tr>
<tr>
<td>APPENDIX A. Source Code Listing</td>
<td>A-1</td>
</tr>
<tr>
<td>APPENDIX B. Results of Runs</td>
<td>B-1</td>
</tr>
</tbody>
</table>
I. INTRODUCTION

One problem that continues to face the designer of computer systems in which various system elements must share common resources is the arbitration of access to these resources. In most systems, this comes down to the sharing of a common bus that connects these resources to each other. The question is: how can access to this resource, the bus, be handled in a "fair" manner? That is, how can one arrange for every potential user of the bus to have adequate access to the bus while preventing any one user from monopolizing it? Also, how can this be done in a way that is the least taxing in terms of hardware and software?

The method put forth in this paper for handling this problem is a conceptually simple solution involving a rotating priority system. In this way, all boards at some point would have exclusive, and in some cases non-interruptible, access to the bus. Because the priority rotates, no one individual user of the bus can monopolize it. This scheme is described in the next section.

Following this discussion of the arbitration algorithm is an extensive treatment of the program used to simulate the action of the arbitration scheme. Finally, there is a discussion of the results of a number of runs using different values for the probabilities of requesting the bus and of completing a bus access.

II. ARBITRATION ALGORITHM DESCRIPTION

The arbitration scheme itself is conceptually very simple. The \( N \) processors, or bus users, are initially assigned priorities of 0 to \( N-1 \), each processor receiving a different priority. While no accesses are made to the bus, or, more accurately, no processor releases the bus, the priorities remain the same. However, when a processor obtains and then releases the bus, all processors have their priorities incremented by one on a modulo \( N \) basis; that is, the processor with priority 0 goes to priority 1, priority \( N-2 \) goes to \( N-1 \), and priority \( N-1 \) goes to 0. In this way, the highest priority processor, the one with the greatest probability of having had access to the bus now receives the lowest priority. A "fairer" method would have the processor releasing the bus receive the lowest priority; however, this would require some method for doing a wholesale reassignment of priorities, whereas the method described requires only incrementing counters on each processor.

However, once a processor receives the bus, it does not necessarily have non-interruptible access to it; the processor may lose the bus to a processor having higher priority. If this happens, the executing process is blocked, the processor relinquishes the bus, the priorities of all the processors are updated, and the requesting processor with the highest priority is granted the bus. The blocked processor now has a pending bus request that will be answered when its priority again exceeds that of all other requesting processors.
III. BUS SIMULATION PROGRAM

In order to develop an understanding as to how this system would perform over time, a program was written that will simulate the performance of this type of scheme over a number of iterations. To allow for a degree of variability, different parameters were set up in the form of probabilities: in particular, the probability of a processor failing, of requesting the bus, and of the processor completing its bus access at any particular time. In this manner, different bus loads could be simulated. The basic structure of the simulation is described below. (See Appendix A for the source code listing.)

The processors are represented in the program as a relatively large data structure containing various flags, probabilities, counters, and value-holders. The flags for each process are: ACTIVE, which shows if the processor is healthy; BLOCKED, which is set if the processor has the bus but has been interrupted; WAITING, which is set if the processor has a bus request which has not been met; and RUNNING, which indicates that the processor has run or is running since the last time update. The probabilities contained in each process structure are those of failure (PROB_FAIL), making a bus request (PROB_REQUEST), and of completing a bus request on any given cycle (PROB_COMPLETE). The counters contain information on the cycles used in the present block, wait, or bus access (PRESENT_BLOCK, PRESENT_WAIT, and PRESENT_RUN), the total number of cycles spent in blocks, waits, or bus accesses (TOTAL_BLOCK, TOTAL_WAIT, NUM_BLOCKS, NUM_WAITS, and NUM_RUNS). The longest of any individual block, wait, or run is also tracked (LONGEST_BLOCK, LONGEST_WAIT, LONGEST_RUN). Finally, and perhaps most important to this simulation, is PRIORITY, which contains the processor's present priority.

To begin the simulation each processor structure has its values initialized through the procedure INIT(). The ACTIVE flag is set, the BLOCKED, WAITING, and RUNNING flags are cleared, the counters are cleared, and using the procedure SET_PRIORITY(), the priority of each processor is randomly set to an integer value between 0 and N-1 (since in this simulation the processors cannot run or request the bus concurrently, they must instead be polled sequentially.) This random setting of the priorities of the different processors helps to nullify any effects that might occur due to having the processors arranged sequentially by priority. After this, the various probabilities are set. The program allows for this to be done randomly with a default value or with the user specifying the probability for each processor individually. These options allow for the tailoring of the bus loading to one's preference.

The main program itself consists mainly of two large loops. The outer loop determines how many times polling of the processors occurs and the inner loop determines how each processor will act when it is polled. At the beginning of the outer loop are print statements that report the status of the system every thousand iterations. At the end of the outer loop is a call to the UPDATE_TIME() procedure which updates the counters within each processor structure, incrementing as necessary the number of cycles in the RUN, WAIT, or BLOCK states. The inner loop is where the actual computations that determine processor state are performed. The routine will check the referenced processor's current state and then, based on this state and the probabilities associated with this processor, the next state of the processor is determined.
At the beginning of this inner loop, the processor under consideration may be in one of two main states, either healthy and functioning (ACTIVE=1) or "dead" (ACTIVE=0), having failed at some point in the past. If the processor is dead, then the program continues on to the next processor. Figures 1 through 3 are flow charts of this portion of the program.

On the other hand, if the processor has not failed, it will be in one of four states as it enters processing. It will either be the present bus master, it will be blocked (it had the bus at one time but was preempted before it could complete the transaction), it will be waiting (the processor has requested the bus, but has not yet received it), or it could have no pending requests. Each of these states requires that different actions be taken and each have different alternatives to choose. Each of these alternatives is chosen through the use of the probabilities assigned to each processor and the actions of a random number generator. Any time the value received from the random number generator is less than the value of the assigned probability, the action associated with that probability will be performed.

The first state to be looked at is that of the processor being the present bus master. This is perhaps the simplest of all the states to process. The first check is to see if the processor fails at this point. If it does, then its ACTIVE flag is cleared, BUS_MASTER and PRIORITY are set to FREE, and UPDATE_PRIORITY() is called. If the processor remains active, then RUN_PROCESS() is called to see if the processor will complete its access this cycle. The program then continues on to the next processor.

RUN_PROCESS() is a very simple routine itself. If the PROB_COMPLETE associated with this process is greater than some random number, then the bus is freed again. UPDATE_PRIORITIES() is then called.

With the next three states, BLOCKED, WAITING, and NO_PENDING_REQUESTS, again the possibility of processor failure is checked first, and, with these three, the processing is the same. If the processor does fail, ACTIVE is cleared, LONGEST_WAIT, LONGEST_BLOCK, TOTAL_WAIT, and TOTAL_BLOCK are all updated with respect to PRESENT_WAIT and PRESENT_BLOCK. The flags WAITING and BLOCKED are then cleared.

The next check made on processors in one of these three states is whether or not it will make a request for the bus. For the BLOCKED and WAITING states this is automatic. For the NO_PENDING_REQUESTS state, this is checked through its PROB_REQUEST value. If there is no request, the program goes on to the next processor.

Next, having made a request for the bus, the priority of the requester is compared to that of the present bus master, that is, to the value stored in the global variable PRIORITY. If the priority of the requester is the lesser of the two values, then the program continues on with the next processor; otherwise, the requesting processor becomes the new bus master. If the bus is free, then this is accomplished by writing the processor's number into BUS_MASTER, its present priority into PRIORITY, and its BLOCKED and WAITING flags cleared. After this, RUN_PROCESS() is called to see if it will complete its access in this cycle. The program then continues on to the next processor.
If, on the other hand, the bus is not free when the requester receives the bus, the old bus master must be blocked. This is accomplished by setting the bus master's BLOCKED flag, incrementing its NUM BLOCKS, and calling UPDATE PRIORITIES(). The requester is then given control of the bus as in the previous paragraph.

After the program has stepped through all N of the processors, it is considered to have completed one BUS CYCLE. At the end of each bus cycle, the routine UPDATE TIME() is called in order to keep track of the amount of time spent by each processor, either on the bus, waiting for initial access to the bus, or in a blocked state. The routine looks at each processor data structure, looking at what flags are set, and updating the appropriate times.

First, it looks at the RUNNING flag. If it is set and the processor is the present bus master, then PRESENT RUN is incremented and LONGEST RUN is updated. If it is not the bus master, then PRESENT RUN is cleared. In any case, the RUNNING flag is cleared. Next, UPDATE TIME() checks the BLOCKED and WAITING flags and updates either PRESENT WAIT or PRESENT BLOCK, respectively. Finally, the routine sets the RUNNING flag of the PRESENT BUS master.

At this point, there may exist some confusion over why UPDATE TIME() clears and then sets the RUNNING flag while the other flags are left alone. The reason is that, with the structure of the simulation, once a processor is put on WAIT or BLOCK, it remains there until the next bus cycle. On the other hand, if it is running, it may be blocked any time during the present cycle or some future cycle. In order to keep track of the amount of time some processor had the bus, it is assumed that the processor keeps the bus for one entire cycle if it had the bus for any portion of that cycle. For this reason, when a processor is blocked, the RUNNING flag is not reset until UPDATE TIME() is called at the end of the cycle. In this way, all the processors that ran during the cycle get credit for that cycle, and by setting the bus master's RUNNING flag again, it will get credit the next time the update routine is called at the end of the next cycle.

After the outer loop has run the required number of iterations (10,000 in this simulation), the results are tallied. The total amount of time spent running in a WAIT or in a BLOCK is tallied, as well as the total number of times processors were put in a RUN, WAIT, or BLOCK. From these, average times may be computed. Also reported are the longest times in any state.
IV. SIMULATION RESULTS

Appendix B gives the results of a number of runs. The first three use random values for the probabilities of failure, request, and completion. (The intermediate results from the first run are presented in order to give the reader an idea of what is displayed during a run.) As can be seen from these, although the total times and longest times spend in a WAIT or BLOCK state have considerable variation from one run to another, the average time in a WAIT or BLOCK remains consistently small, generally less than 10 cycles. In addition, the total number of cycles in which the bus was active remained relatively constant.

The rest of the runs presented represent situations that range from a very heavily loaded bus (a high probability of requesting the bus along with a low probability of completing an access) to a very lightly loaded bus (a low probability of request with a high probability of completion.) As can be seen on the heavy loading run, although the number of cycles the bus is in use increases by about half over the random loading, the amount of time in a BLOCK or WAIT more than doubles. However, the average time spent in a BLOCK or WAIT is still under 10 cycles - the same as with the random case. The next case, high probability for both request and completion, is perhaps the ideal bus loading situation. Bus utilization is more than doubled over any other case. In addition, the average WAIT and BLOCK is down to one.

The case in which both the probabilities are low (a request is not likely, but if there is one, it is not likely to complete on any given cycle) is about average when compared to the other cases. The average delays are in the same range as all the others, under 10 cycles, and because there are fewer requests, the bus is used less often than the higher request situations, even the one with the high blocking rate. The final situation, with low request probability and high completion probability, has the lowest rate of blocking and waiting, though not much lower than the high request/high complete. Again, because the request rate is low, bus utilization is dropped.

V. CONCLUSIONS

This system of bus arbitration using a rotating priority scheme appears to be an effective and efficient method of handling bus arbitration. Average delays are consistently low in all cases tested; failure of individual system elements appears to have little effect on overall system performance; and the conceptual simplicity of the system should allow for relatively efficient implementations in hardware.
Figure 1. Flowchart for inner processing loop.
Figure 2. Flowchart of bus master processing loop.
Figure 3. Flowchart of other (waiting, blocked, no request) processing loop.
/* global.h - Defines such things as the structure for the process */

struct Process
{
    char Active; /* flag showing whether or not the processor */
    char Running; /* has run during this cycle */
    char interrupted; /* has been interrupted */
    int received; /* has received a bus request */
    int Priority; /* present priority of processor */
    float ProbConnected; /* the probability that a bus request will be */
    float ProbComplete; /* the probability that if the processor has */
    int PresentRun; /* the number of cycles waiting on present request */
    int PresentBlock; /* the number of cycles the processor spent to wait */
    int PresentTotal; /* the number of cycles in a blocked state */
    float PresentTotalCycles; /* the longest time blocked */
    int PresentRun; /* the number of non-interrupted cycles possessing */
    int LongestRun; /* the longest time spent in any single run */
};

/* End of global.h */

/* ProcArray[NPROCESSORS]; */

/* Random() - function to produce a random number x, 0 <= x < 1. */
I.

```
/* Routine to produce the random numbers */
seed = (int)^2L + seed + (INT)(x - (INT));
return ((INT)random()/(float)maxint);
*/
/* end Random */
```

---

```
/**
UpdateTime() - Update the wait times on all the processors
**/
```

---

```
Line# Source Line Microsoft C Compiler Version 4.00
10 void UpdateTime()
{ short int i;
  for (i=0; i<PROCESSORS; i++)
    if (Priority[i].Running)
      if (BusyMaster == 1)
        Process[i].Running = 0;
      else
        Process[i].Running = Process[i].Length;
    Update the time on each blocked or waiting process */
    if (Priority[i].Waiting)
      Process[i].Waiting = Process[i].Length;
    else
      Process[i].Waiting = 0;
    if (BusyMaster == FREE)
      Process[i].Running = 0;
  } /* End of UpdateTime() */
```

---

```
SetPriority() - Do random initialization of priorities
```

---

```
Line# Source Line Microsoft C Compiler Version 4.00
10 void SetPriority()
{ short int i;
  for (i=1; i<PROCESSORS; i++)
    if (BusyMaster == FREE)
      for (i=1; i<PROCESSORS; i++)
        do
          Process[i].Priority = int(random());
        while (Process[i].Length = 0);
  } /* End of SetPriority() */
```
```c
Name          Class Offset Register
-------------- ------- -------

UpdatePriority() - Update the priorities on all the processors

void UpdatePriority()
    short int i;
    for (i=0; i<NUMPROCESSORS; i++)
        /* To update the priority of each processor, increment each and
         * end by NUMPROCESSORS */
        ProcArray[i].Priority = (++ProcArray[i].Priority) % NUMPROCESSORS;
    
} /* End of UpdatePriority() */

RunProcess() - Routine to see if bus access will complete this cycle

void RunProcess(ProbComplete)
    {test ProbComplete;
        if (ProbComplete) = ProbComplete
            bus better complete access. Release the bus and
            update the priorities */
            procArray[i].Busy = 0;

    } /* End of RunProcess */

init() - Routine to initialize variables for computation

init()
    short int; /* Need random number generator */
    unsigned int; /* (call for Get Time */
    long int; /* (call for Get Time */
    int; /* Set the initial priorities for all of the processors */

for (i=0; i<NUMPROCESSORS; i++)
    ProcArray[i].Priority = 0;
```

Microsoft C Compiler Version 4.00
```c
#define NUMPROCESSORS 8
#define PROBA

// Set the values for the various probabilities */

/* ProbFail */

pick3: printf(<<the probabilities of failing

Print(02, 0.0001 * Random); /*

Let the values be the random number from

all failing during the first few iterations.

printf("Probability of failure Xd \%v\%, ProcArray[i].ProbFail\);)

/*

Print(01, (default value)),

for (i = 0; i < NUMPROCESSORS; ++)

ProcArray[i].ProbFail = Value;

/*

Print(01, (probability of failure for processor Xd ",",));

ProcArray[i].ProbFail = Value;

Print("\n\n);

/*

Print(01, (probability of request for processor Xd ",",));

ProcArray[i].ProbRequest = Value;

Print("\n\n");

Print("\n\n");

/*

Print(01, (probabilities of trying again

Try again.\n\n");

Print("\n\n");

/*

Print(01, (probability of completing a bus access

Print("\n\n");

Print("\n\n");

/jk*/
```

```c
for(i=0; i<NUM_PROCESSORS; i++)
    ProcArray[i].ProbComplete = Random();
printf("Probability of completion for processor
X\nProcArray[i].ProbComplete = Multiplier%1d\n",i);
for(j = 0; j < NUM_PROCESSORS; j++)
    ProcArray[j].ProbComplete = Value;
for(i = 0; i < NUM_PROCESSORS; i++)
    ProcArray[i].ProbComplete = Value;
print("probability of completion for processor Xd: "\nProcArray[i].ProbComplete = Value;
getInput();
printf("Please retry. Very dumb. Try again\nProcArray[i].ProbComplete = Value;
/* End of Init */

Local Symbols
Name            Class  Offset  Register
in              auto    -0020  of
tot              auto    -0033  of
proc              auto    -0000  of

printf("\nProcArray[i].ProbComplete = Value;
getInput();
printf("Please retry. Very dumb. Try again\nProcArray[i].ProbComplete = Value;
/* End of Init */

Local Symbols
Name            Class  Offset  Register
in              auto    -0020  of
tot              auto    -0033  of
proc              auto    -0000  of

if (ProcArray[i].Active) // Processor is dead. Do nothing */
    continue;
    if (i == Master)
        bank as dead
        Unload Master
        Complete Priorities
        Update Priorities
        Update Time
    else
        if (Random() < ProcArray[i].ProbFall)
            The processor just failed. Bank as dead. Release
            bank. Update the Master to PROC and priority
            to PRI. Update all of the priorities */
            printf("Processor Xd failed at time Xd, \nProcArray[i].Active = 0;
            Master = XXX;
            Priority = YYY;
            continue;
        
/* End of Bus Master failure */
/* Check to see if processor will complete during

A-5
```
this bus cycle. Else continue to hold bus. */

} /* end of process if processor already has bus */

/* II. Does not have bus. */

B. Black

1. If blocked or waiting

a. Update Total/LargestBlock/Wait

B. Does not have bus.

1. If not blocked or waiting.

a. Set time.

b. Update time.

2. No requests.

a. Nothing.

b. If (Random) < ProcArray[j].ProbFail)

"The processor just failed. Mark as dead. */

ProcArray[j].Active = 0;

1. If ProcArray[j].Blocked || ProcArray[j].Waiting

b. Use blocked/waiting flags to clear blocks and clear


if ProcArray[j].LargestLock + ProcArray[j].PresentLock;

ProcArray[j].LargestLock = 0;

ProcArray[j].PresentLock = 0;

ProcArray[j].Waiting = 0;

C. else do nothing */

/" end of failed processor */

/* Else it did not die. Check and see if it is waiting.

No requests for bus. See if it will get it */

if (ProcArray[j].Priority > Priority)

/* (Priority in range.

B. Block and replace old

C. */
if (ProcArray[i].Blocked && ProcArray[i].Waiting) {
    ProcArray[i].Waiting = false;
    for (j = 0; j < NUMPROCESSES; j++)
        if (ProcArray[j].NumBlocks > ProcArray[i].NumBlocks)
            ProcArray[j].NumBlocks -= 1;
        else if (ProcArray[j].NumBlocks == ProcArray[i].NumBlocks)
            ProcArray[j].NumBlocks--;  
        ProcArray[i].NumBlocks = 0;
    ProcArray[i].ProcComplete = true;
    ProcArray[i].Rank = 0;
    ProcArray[i].ProcState = ProcArray[i].Processors;
}

// and else bus is free */
// else will not get bus. Put on wait if not
// blocked or waiting already. Update times.
/* End main */

<table>
<thead>
<tr>
<th>Name</th>
<th>Class</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>base</td>
<td>-0000</td>
</tr>
<tr>
<td></td>
<td>base</td>
<td>-0000</td>
</tr>
</tbody>
</table>

Global Symbols

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size</th>
<th>Class</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avail blocks</td>
<td>struct/ary</td>
<td>54</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Num free blocks</td>
<td>int</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Num of blocks</td>
<td>int</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Num of free blocks</td>
<td>int</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Avail list</td>
<td>struct/ary</td>
<td>54</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Num free list</td>
<td>int</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Num of list</td>
<td>int</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Num of free list</td>
<td>int</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Microsoft C Compiler Version 4.00

Code size = 0x10 (256)
Data size = 0x00 (0)
Heap size = 0x00 (0)
No errors detected
APPENDIX B

RESULTS OF RUNS
A. Run 1: Random Loading

Do you wish to set the probabilities of failure

<table>
<thead>
<tr>
<th>Probability</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Do you wish to set the probabilities of making a bus request

<table>
<thead>
<tr>
<th>Probability</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Do you wish to set the probabilities of completing a bus access

<table>
<thead>
<tr>
<th>Probability</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B-1
Do you wish to set the probabilities of making a bus request

Do you wish to set the probabilities of completing a bus access

C. Run 3: Random Loading

Do you wish to set the probabilities of failure

Do you wish to set the probabilities of making a bus request

Do you wish to set the probabilities of completing a bus access
D. Run 4: High Probability of Request/Low Probability of Completion

Do you wish to set the probabilities of failure
- individually, or
- default value?

Do you wish to set the probabilities of making a bus request
- individually, or
- default value?

Do you wish to set the probabilities of completing a bus access
- individually, or
- default value?

Total number of cycles blocked: 59045
Total number of cycles in wait: 20269
Average time for a block: 706
Average time in a block: 16
Average time in a run: 63
Number of blocks for a processor: 1692
Number of runs for a processor: 3570

E. Run 5: High Probability of Request/High Probability of Completion

Do you wish to set the probabilities of failure
- individually, or
- default value?

Do you wish to set the probabilities of making a bus request
- individually, or
- default value?

Do you wish to set the probabilities of completing a bus access
- individually, or
- default value?

Total number of cycles blocked: 110088
Total number of cycles in wait: 58088
Average time for a block: 8
Average time in a block: 1
Average time in a run: 62
Number of blocks for a processor: 1579
Number of runs for a processor: 5270
F. Run 6: Low Probability of Request/Low Probability of Completion

Do you wish to set the probabilities of failure

- individually
- default value

Do you wish to set the probabilities of making a bus request

- individually
- default value

Do you wish to set the probabilities of completing a bus access

- individually
- default value

Total number of cycles blocked: 4256
Total number of cycles in bus: 14677
Mean time for a block: 86
Mean time in a block: 9
Average time in a block: 1053
Mean number of waits for a processor: 124
Mean number of runs for a processor: 138

G. Run 7: Low Probability of Request/High Probability of Completion

Do you wish to set the probabilities of failure

- individually
- default value

Do you wish to set the probabilities of making a bus request

- individually
- default value

Do you wish to set the probabilities of completing a bus access

- individually
- default value

Total number of cycles blocked: 1236
Total number of cycles in bus: 2720
Mean time for a block: 9
Mean time in a block: 0
Average time in a block: 0
Mean number of waits for a processor: 274
Mean number of runs for a processor: 219

B-5/(B-6 blank)
### DISTRIBUTION

<table>
<thead>
<tr>
<th>Organization</th>
<th>Addressee</th>
<th>Mailing Address</th>
<th>City, State ZIP</th>
<th>Number of Copies</th>
</tr>
</thead>
<tbody>
<tr>
<td>US Army Materiel System Analysis Activity</td>
<td>ATTN: AMXSY-MP</td>
<td>Aberdeen Proving Ground, MD 21005</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>IIT Research Institute</td>
<td>ATTN: GACIAC</td>
<td>10 W. 35th Street</td>
<td>Chicago, IL 60616</td>
<td>1</td>
</tr>
<tr>
<td>AMSMI-RD, Dr. McCorkle</td>
<td>Dr. Rhoades</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>-RD-AS-SS, Mr. Sims</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>-RD-BA</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>-RD-GC, Mr. Sliz</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>-RD-GC-S</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>-RD-CS-R</td>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>-RD-CS-T</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>-GC-IP</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>AMCPM-ML, Mr. Yarbrough</td>
<td>-ML-TM, Mr. Crosswhite</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>-ML-TM-R, Mr. Hill</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

DIST-1/(DIST-2 blank)