AN EXPERIMENT USING MODULAR PROGRAMMING METHODOLOGY

Dissertation submitted to the Jawaharlal
Nehru University in partial fulfilment
for the award of the degree of
MASTER OF PHILOSOPHY

A
ORUGNTI VENU GOPALA KRISHNA

School of Computer and System Sciences
Jawaharlal Nehru University
New Delhi-110 067

1982

C E R T I F I C A T E

This dissertation entitled 'An Experiment Using Modular Programming Methodology' embodies work carried out at the School of Computer & Systems Sciences, Jawaharlal Nehru University, New Delhi-110067.

This work is original and has not been submitted in part or full for any other degree or diploma of any University.

O.V.G. Krishna
(O.V.G. KRISHNA)
STUDENT

N.P. Mukherjee
PROF. N.P. MUKHERJEE
DEAN

R. Sadananda
DR. R. SADANANDA
SUPERVISOR

C O N T E N T S

ACKNOWLEDGEMENT

I gratefully acknowledge the tremendous and unimaginable co-operation of my supervisor, Dr. R.Sadananda had given me throughout my stay at National Centre for Software Development and Computing Techniques (NCSDCT) - Bombay and in the preparation of the dissertation, in a very short period. His mountainous backing let me persue my work at NCSDCT, unhindered. I express my sincere thanks and gratitude to him for his suggestions, encouragement and co-operation.

"You should never have timid attitude towards studies..jump into it and you will succeed". Thus says Sri V.R. Prasad, Scientific Officer, N.C.S.D.C.T. whose influence induced the spirit of work in me and made me really enjoy my work. My stay at NCSDCT with him laid the foundation for my hopefully bright future in computer science. Without his constant technical co-operation I could never have succeeded in doing this work. I express my debt of gratitude and sincere thanks to him.

I thank Prof. N.P. Mukherjee, the Dean of School of Computer and Systems Sciences of this University for allowing me to work at an outside institution. I also thank Prof. R. Narasimhan, the Director of N.C.S.D.C.T. for providing me with all the necessary facilities to do my dissertation work.

My special thanks are due to my pals at I.I.T. Bombay and J.N.U., New Delhi for getting me the computer printouts in time.

Finally, I thank Mr. K. Chand for typing this dissertation in a very short period.

(O.V.G. KRISHNA)

AUTOMATIC COMPUTERS HAVE NOW BEEN WITH US FOR MORE THAN A QUARTER CENTURY. THEY HAVE HAD A GREAT IMPACT ON OUR SOCIETY IN THEIR CAPACITY AS TOOLS, BUT IN THAT CAPACITY THEIR INFLUENCE WILL BE BUT A RIPPLE ON THE SURFACE OF OUR CULTURE COMPARED WITH THE MUCH MORE PROFOUND INFLUENCE THEY WILL HAVE IN THE CULTURAL HISTORY OF MANKIND.

- the humble programmer

ORGANIZATION OF THE DISSERTATION

The organization of the dissertation and contents are as follows:

Firstly , in introductory chapter the statement of the aim of the dissertation and methodology chosen to achieve the task, we aimed at, are mentioned. A brief review of some examples in the literature relevant to the work we are attempting is presented here.

In chapter I, the simulation process of TDC 316 is discussed in detail. The design is presented in step-wise refinement method.

Chapter II is devoted for the study of 'modularization methodology' which is used in developing the program. The modularization methodology is studied in the light of two concepts, namely the structure of the modular programs and the criteria to be used in decomposing the large system into modules. Simultaneously, our modular simulator is analyzed in the light of the above concepts and explanations for following any structure or criteria in the decomposition are presented.

Chapter III deals with the design and the developmental aspects of the simulator SIM16. In this

chapter the problems that are encountered during the design and development process of the program and also how we have overcome them,are presented. In the second section of this chapter a brief discussion about how our modular simulator SIM16 can be implemented for other systems or configurations is given.

Next chapter is concluding chapter which briefly summarizes the contents of previous chapters and also offers some conclusions, comments and suggestions on the simulator program.

Appendix 1 contains a very brief note on the modular features of the language which is used for writing SIM16, namely CCNPASCAL.

Appendix 2 contains the commands to use SIM16 and also listing of the addresses of TDC 316 G.P.R s and device registers, for understanding of the constants used in the simulator.

Appendix 3 contains the listing of SIM16 program alongwith the CONTEXT files of each module.

INTRODUCTION

Objective of the dissertation is to carryout an experiment using modular programming methodology with a particular goal of developing an easily modifiable software system.

The need to carryout such an experiment is that the software systems(compilers, assemblers, interpreters, simulators, commercial data processing programs etc.) are generally developed as special purpose systems which do not allow easy modifications after their development.

Modular Programming is a programming methodology by which a large and complex task can be divided into small, easy and "intellectually manageable" tasks.

As an instance,we have taken the task of modularizing a simulator for computer system TDC316 which was already developed as a big, single monolithic program. Modularization of this system is carried out to make it more 'flexible' i.e. to facilitate its easy implementation to a number of different configurations/computer systems.

Simulation, in general usage, is defined as an act or process that gives the appearance or effect of some part

of reality. This definition is too broad. A more precise definition of simulation is as follows (Maisel Harbert, 19):

'Simulation is a technique for conducting experiments on a digital computer; this involves certain types of mathematical and logical models that describe the behaviour of business, economic, social, biological, physical or chemical systems (or some component thereof) over periods of time'.

For our purpose the simulation of a computer system means making the host machine behave like the machine being simulated.

'Software Engineering' is a relatively new name coined to denote a rapidly growing body of knowledge with computer-program design, composition and production. Software Engineering is equally concerned with the quality of the programmers product and with the efficiency of the process of getting this product (W.M. Turski, 21).

"Software flexibility" is one of the concepts of Software Engineering. Flexibility of Software is of two types

'Adaptability' and 'Portability' (Buxton et al, 22, pp 166).

Adaptability is concerned with enabling a given system to grow and change.

Portability is the more restricted problem of area of moving a specific software system from one environment to another.

The concept underlying our objective, namely, 'Modifiability' is related closely to adaptability.

There exist a number of examples for software which is sufficiently portable or adaptable. An early example is the ALCOR ILLINOIS compiler for ALGOL 60, which was built for an IBM 7090 and was transferred by David Gries to an IBM 7044 in just two-man weeks. However, it is to be noted that there is little architectural difference between these two machines. Its portability was achieved mainly through parameterization (D. Gries et al, 23).

Another experiement is of P.C. Poole and W.M. Waite, using a 'mobile programming system' with the macro processor STAGE2 as tool. (Poole and Waite, 24, 25). This STAGE2 itself is highly portable and has been implemented on twenty

different computer systems requiring about one man-week of efforts to obtain a running version in each case. STAGE2 was coded in the assembly language of a special purpose computer called FLUB, which was designed to handle the data structures relevant to micro processing: trees, strings and integers. STAGE2 was implemented via SIMCMP, a simple compiler developed by the same persons (Richard J. Orgas and W.M. Waite, 24).

There are many more interesting approaches scattered in the literature.

Another example of highly portable software system is the CCNPASCAL compiler initially developed for DEC system 10 at NCSDCT-TIFR-Bombay (V.R. Prasad, 15). CCNPASCAL compiler is a self-compiling modular compiler. The parser is separated from the code generator and performs considerable machine independent optamization. To adapt the compiler to a new machine mostly the code generator has to be modified requiring an effort of about 3-6 man-months. CCNPASCAL is implemented for a number of systems such as DEC 10, PDP 11, MOTOROLA 68000, INTEL 8085 and TDC 316.

In all the examples quoted above one common thing to

be noted is that the software system is 'moved' to different target machines (portable).

Our objective is also to develop a highly modifiable software system. There are no drastic conceptual differences between the example problem we have chosen to carryout the experiement and the objective of the above examples. However, in developing our software system, we have not considered the portability criterion. The modular simulator will work only on the machine on which we have developed it initially; but it will be adaptable to different changes - growths and contractions. The simulator can be made to work for a number of computer systems with reduced effort. This requires that the inner details and the logic of the simulated machine dependent parts may be separated from the overall structure and logic of the program and the simulated machine independent parts.

An example, which is similar to the one we are attempting, is the development of modular simulator for the Fairchild F8 microprocessor by Kallol K. Bagchi et al of Jadavpur University, India. This simulator consists of three basic modules, namely, microprocessor simulator which

simulates the F8 microprocessor, I/O process simulator which simulates DMA activity and External Device Simulator which simulates a vide range of synchronous and asynchronous devices. By shadowing completely or partially a particular module, various micro computer configurations may be derived. In addition to the three basic modules, the package also contains a cross-assembler including macro facilities (Kallor K. Bagchi et al, 26).

Another software system which is 'adaptable' to the different modifications is the KWIC index production system. This system design is presented in both conventional and unconventional decomposition of the [illegible] systems into modules and then the two decompositions are compared. Through this program modularization, Parnas D.L. suggests a number of criteria to be used in decomposing the system into modules (Parnas D.L., 1). These criteria are followed to a good extent in developing our modular simulator.

As a whole, though our experiment does not have major conceptual differences with the examples existing already in the literature, it is still worth carrying out; because modularization can be done in a number of ways depending upon

the logic of the program and the goals of the work. Although certain well-established criteria are to be followed in decomposing the systems into modules, a practical experience with modularization concept gives a deep insight into its process. Also the modularization in CCNPASCAL gives a good picture of the already popular and efficient methodology of the Software Engineering.

CHAPTER I

SIMULATION OF THE COMPUTER SYSTEM TDC 316

The simulation of computer systems usually are carried out to evaluate systems performance effectively.

In a comprehensive article on systems evaluation (Lucas, 20), it is noted that simulation is 'the most potentially powerful and flexible of the evaluation techniques' and 'is the most adequate for all purposes of evaluation', but the greatest drawback of simulation is its relatively high cost.

We shall not go into the details of simulation and its advantages here, because our objective is to modularize an already existing simulator for computer system TDC 316 to make it more 'flexible'. However, for our purpose, we shall briefly discuss the simulation process of any computor system in general without going into the details of any particular system. Lateron we will discuss the simulation of TDC 316 in particular.

Simulation process of any arbitrary computer system has the following tasks to be performed. There may be slight deviations from this general process for simulation, but more or less it is general to all computer systems.

The simulation process is outlined as consisting of a number of subtasks. A detailed description of simulation of TDC 316 system is given at later chapters. Here it is intended to give just a birds-eye-view of the simulation process to justify our usage of the methodology namely MODLARIZATION AND SEPARATE COMPILATION (V.R.Prasad, 17).

INITIATION:

Initiates a dialogue with the interactive user:

1. to initialize the system
2. to start the simulation process
3. to provide online debugging facility (via inserting the break points for instructions and address, and accessing required memory locations and modifying their contents and so on)
4. to provide the simulation summary at the end of simulation process.

MEMORY ACCESS:

A. Declare different data structures and define routines necessory for accessing the memory blocks of: core memory, central processor and

I/O area.

B. Define miscellaneous utility routines to perform various operations such as complementing a word, reading from or writing into a particular portion of a memory word (e.g.; read bit no.6 of status register, get left word of the data and so on).

REALTIME TASKS:

A. Create and maintain event and interrupt schedules.

B. Do interrupt, event and trap actions.

DEVICE SIMULATION:

Simulate the required devices such as Key board, Tele Type Printer, High speed Reader, Clock and Plotter and so on.

PREPARATION FOR EXECUTION:

Fetch the instruction from memory. Decode the instruction to find OPCODE, MODE of operation etc; Evaluate the source and destination operand address.

INSTRUCTION EXECUTION

Get the specified source and destination operands from appropriate blocks of memory (by using the routines defined in the step number 2) perform the actions required such as adding, multiplying shifting loading jumping from one location to another and so on. Serve all interrupts waiting in the queue.

These are the general steps to simulate any computer system.

From the above description it is worth noting some interresting points.

Firstly, the whole task of simulation has been divided into a number of sub tasks such as INITIATION, MEMORY ACCESS, DEVICE SIMULATION, PREPARATION FOR EXECUTION, INSTRUCTION EXECUTION. These sub tasks can be made, by choosing proper criteria to decompose them into sub tasks, more or less independent of others. Therefore, they can be put in different modules and also can be developed by different persons. All that one needs to know, to develope another module, is how to use the routines and data in the other modules. One need not bother much about the details

of the code body. Therefore these modules can be separated out from each other.

There are other interesting points worth noting down. Some of the modules are mechine/configurration independent such as the INITIATION, part B in the MEMORY ACCESS module, and part A of REALTIME TASKS. That means when writing simulator for two different systems/configurrations these tasks have to be repeated almost exactly. Therefore, these parts can be separated from others. Then, to simulate another system/configurration keep these parts intact and replace (or modify) only other parts. One more point is that we kept the simulation of the devices in a separate module. Therefore, if at a later stage one wants to simulate one more device, say, Plotter, he can just add one routine to this module without changing the rest of the Simulator.

For the above said reasons and also for the reasons which will become clear in later sections the method of MODULARIZATION and SEPARATE COMPILATION is chosen to develope a sort of generalised simulator.

We will discuss the simulation of TDC 316 (ECIL 1971)

computer system in detail. The simulator, we call it, SIM16, was intended to:

(a) execute machine language programs of TDC 316

(b) simulate a realtime environment for TDC 316

(c) provide on-line debugging facility.

SIM16 is written in a language called CCNPASCAL (V.R. Prasad, 15). This is a PASCAL kind of language with some additional features to support:

Modular and Concurrent programming

Type parameterization

Data abstraction facilities.

A discussion on modular features of CCNPASCAL is given in later chapters.

Input to SIM16 is a machine language program of TDC 316 and output is the result of the program which is supposed to run on TDC 316 system.

We will now present the algorithm for simulation of TDC 316 computer system in step wise refinement method. Later, we decompose it into modules gradually. See ALG.1. (Mathai Joseph, 16).

```
begin SIMULATION
   PERFORM TDC316 SIMULATION
end SIMULATION;
```

```
begin SIMULATION
   DECLARE AND INITIALIZE DATA;
    repeat
      SIMULATE TDC316 SIMULATION
    until END OF SIMULATION
end SIMULATION;
```

```
begin SIMULATION
   DECLARE AND INITIALIZE DATA;
   repeat
      DO REALTIME TASKS;
      if (not SIMULATION ERROR)
      then
         begin
            SIMULATE INSTRUCTION EXECUTION;
            while (TRAP CONDITIONS) do
               DO TRAP ACTIONS
         end
    until (HALT CONDITION or SIMULATION ERROR)
 end SIMULATION;
```

ALG. 1

Version 3 gives a brief algorithm of the simulation process. We will discuss the steps involved in this process in their respective modules.

In the first step we declare and initialize various simulation data. This typically consists of

1. The boolean variables which control simulation process such as
   - halt condition
   - simulation error
   - event schedule empty
   - interrupt schedule empty
   - wait condition
   - trap conditions

   and numeric data such as
   - real time
   - priorities
   - interrupt and event lists
   - status and buffer registers of devices
   - general purpose registers
   - simulated memory
   - trap vectors, base address and so on.

As we go further, we split this data into different modules wherever it is appropriate to keep them.

In the second step real time tasks are performed. The real time tasks are those that creats a real environment of TDC 316 system on the host system for simulation. This part necessiates the capability of keeping track of various events in the system,caused by activities concurrent with the instruction execution process. Some of these events cause 'interrupt request' and might take system into trap state.

In the next step actual execution of machine instruction is performed, by evaluation the addresses of source and destination operands, OPCODE s, MODE of operation and so on. Thus, we can divide the simulation process into two modules consisting of realtime task simulation and instruction execution simulation as shown in the following figure. See Fig. 1.

REAL TIME TASKS

Events:

The real time environment of the TDC 316 can be characterized by the number of concurrent activities in

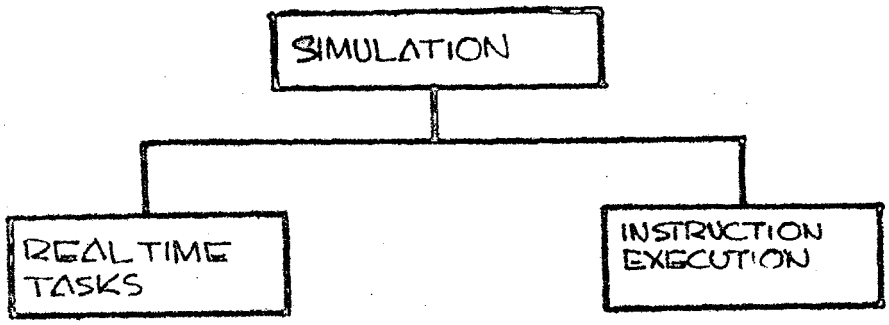
SIMULATION
REALTIME TASKS
INSTRUCTION EXECUTION

FIG.1

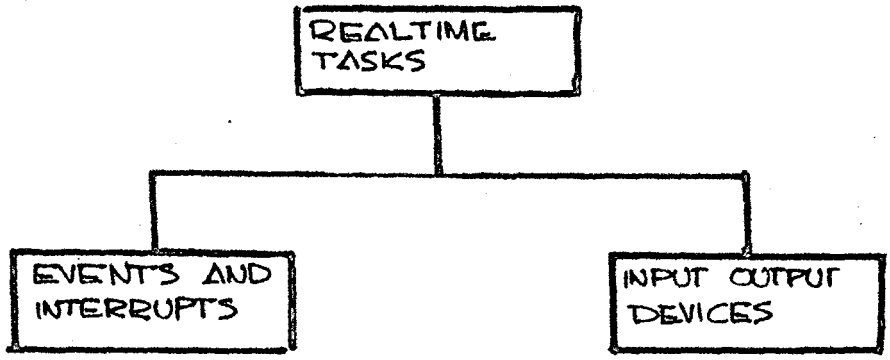
REALTIME TASKS
EVENTS AND INTERRUPTS
INPUT OUTPUT DEVICES

FIG.2

progress in the system at any given time. Some of these activities culminate in predictable EVENTS and hence can also be characterized by the events they cause. As these events occur in real time, a linear list of event times and event types can be used to portray the parallel activities in progress in the system. The assumption that two events will not occur at the same time may be noted. In other words, units of time are fine enough to distinguish between separate events (a restriction imposed by this scheme is that, if separate events do occur at the same; time, the events should be independent of each other's influence).

To illustrate such a concurrent activity and its associated event we will present an example here.

Consider a 300 ch/sec paper tape reader. The initiation of the reader to read a character causes the start of a parallel activity (parallel to the instruction execution process and any other device in operation). 3.33 m.sec later, the reader would have finished reading a character. The reader which was in BUSY state so far, become FREE. The buffer register has the read character and the status of the buffer is FULL. The simulation of

the real time environment requires that at the time of this event, the status of thereader should portray all the above information (by setting or clearing the appropriate bit positions in the status register) and the reader buffer should contain the pertinent character. For values of real time before this event, this should not be the case; and for values of real time after the event, this condition should hold (until some operation is performed which changes the reader status).

INTERRUPTS:

It was stated above that only some of the parallel activities terminate in predictable events in real time. There are other parallel activities for which it is not possible to predict the time of their termination. However, these parallel activities terminate upon the occurence of an allowable state of system.

For instance, consider the waiting queue of interrupt requests. This queue is organized according to decreasing priority so that highest priority interrupt request gets the first attention. However, it is not possible to predict 'when' this interrupt will receive

attention, because the receiving of interrupt request depends on the priority of the c.p.u.

From the above discussion it may be noted that the real time, situation can be modelled by maintaining an event schedule and perserving the queue of waiting interrupt requests.

Thus, we can further catagorize the real time tasks as simulation of event and interrupt occurrences and simulation of input output devices. See ALG 2, Fig.2.

SIMULATION OF EVENT AND INTERRUPTS

This part consists of implementation of event and interrupt schedules and their maintenance. In this part we declare the status registers and routines which do the event and interrupt actions.

We have two ordered lists or queues. Each list or queue has its own ordering key. To keep track of the real time situation, it should be possible to introduce new elements into the selected queue or list according to the value of the ordering queue associated with the new element (e.g., priority of interrupt). So also it should be possible to delete the first element of the selected queue. Further

```
begin SIMULATION
   loop
      if TRAP
      then DOTRAPACTIONS;
      SIMULATE DEVICES;
      if (not WAIT CONDITION)
      then
         begin
            SAVE PC AND CPU STATUS;
            EXECUTE INSTRUCTIONS
         end;
      SERVE INTERRUPTS;
      if (SIMULATION ERROR)
      then return
   end loop
end SIMULATION;
```

ALG.2

more, it should be possible to remove an arbitrary element by identifying the device with which this element is associated.

The first of these operations (entering elements) is used to schedule events or log interrupt requests. The second of these operations (removing the first element) is used for performing the actions required by an event or for servicing interrupt requests.

Let's see how interrupts requests are queued. Consider the previous example of paper tape reader once again. Assume that the device capability to request an interrupt is enabled. Now the occurence of an even signalling the end of a read operation in the device, coupled to - 'interrupt request enabled' state can cause an interrupt request to be scheduled by the device.

The priority of the request depends on the priority of the device. In TDC 316, this priority is hardware assigned and there exists a uniqueue priority identifying each device.

We shall now present an algorithm to implement event and interrupt schedules. See ALG 3,4,5 and 6.

SIMULATION OF I/O DEVICES:

Simulation of I/O devices is done by implementing the real time operations of these devices. Each device is allocated a block of address in the address space and is made to perform various transactions by manipulating the contents of allocated address. Hence we need information storing and retreiving routine for each device.

The storing routines detect conditions in the data to be stored with relevance to the address specified and thereby, cause either scheduling of events or cancellation of scheduled events and logged interrupt requests.

The retreiving routines either, just, fetch data from specified address or also perform some operations required by the side effects of an access to a particular address e.g. clearing a buffer register after accessing it.

Through the use of these two routines, programs can operate the devices.

```
begin EVENT
  while (not EVENT SCHEDULE EMPTY) and
        (SIMTIME >= FIRST EVENT TIME) do
  begin
     REMOVE FIRST EVENT FROM THE SCHEDULE;
     DO EVENT ACTIONS
  end
end EVENT;
```

```
begin INTERRUPT
  while (not INTERRUPT SCHEDULE EMPTY) and
        (CPURRIORITY <= FIRST INTERRUPT PRIORITY) do
  begin
     REMOVE FIRST INTERRUPT FROM SCHEDULE;
     DO INTERRUPT ACTIONS
  end
end INTERRUPT;
```

```
begin TRAP
  while (not SIMULATION ERROR) and
        (TRAP CONDITIONS) do
     DO TRAP ACTIONS
end TRAP;
```

ALG. 3.

Interrupts in TDC 316 lead to a nnique location in the interrupt vector depending on the device. So, we need an interrupt routine for each device which will provide the interrupt vector address associated with the device for invoking the interrupt sequence by CPU (and for performing other functions).

The information shared between all these routines for each device is available in:

(a) the device registers located through addresses assigned in the address space of the device and
(b) the two lists, event schedule and interrupt schedule.

We shall now,see how the above tasks can be separated into different modules to increase the understandability and decrease the mutual interaction between any two modules.

TDC 316 has one unique and interesting feature that all its subsystems components are allocated mutually exclusive address blocks in the unibus address space. This means that the CPU does not have the burden of performing specialized operations for achieving input-output

transactions in the system. In other words, manipulation of contents of certain address (allocated to input output devices) indirectly causes the performance of input-output transactions.

In simulating this address space, two procedures were visualized, one for storing information into address space and another for retreiving information from address space. The storage and retrieval routines of the uni address space invoke further store or access routines of particular subsystems depending on the address part of their arguments. In case no subsystem has been allocated the specified address, the request is considered illegal and should cause a TRAP action in the system. Accordingly the appropriate trap flag is turned on for further processing. Therefore, we have divided the DEVICE SIMULATION part into two groups one that belongs to i/o devices second that belongs to registers, core and CPU. See Fig. 3.

The routines which manipulate status registers of the devices and I/O area are pushed into one module which also contains event and interrupt simulation routines. The routines which manipulate the g.p.r s, c.p.u. and core

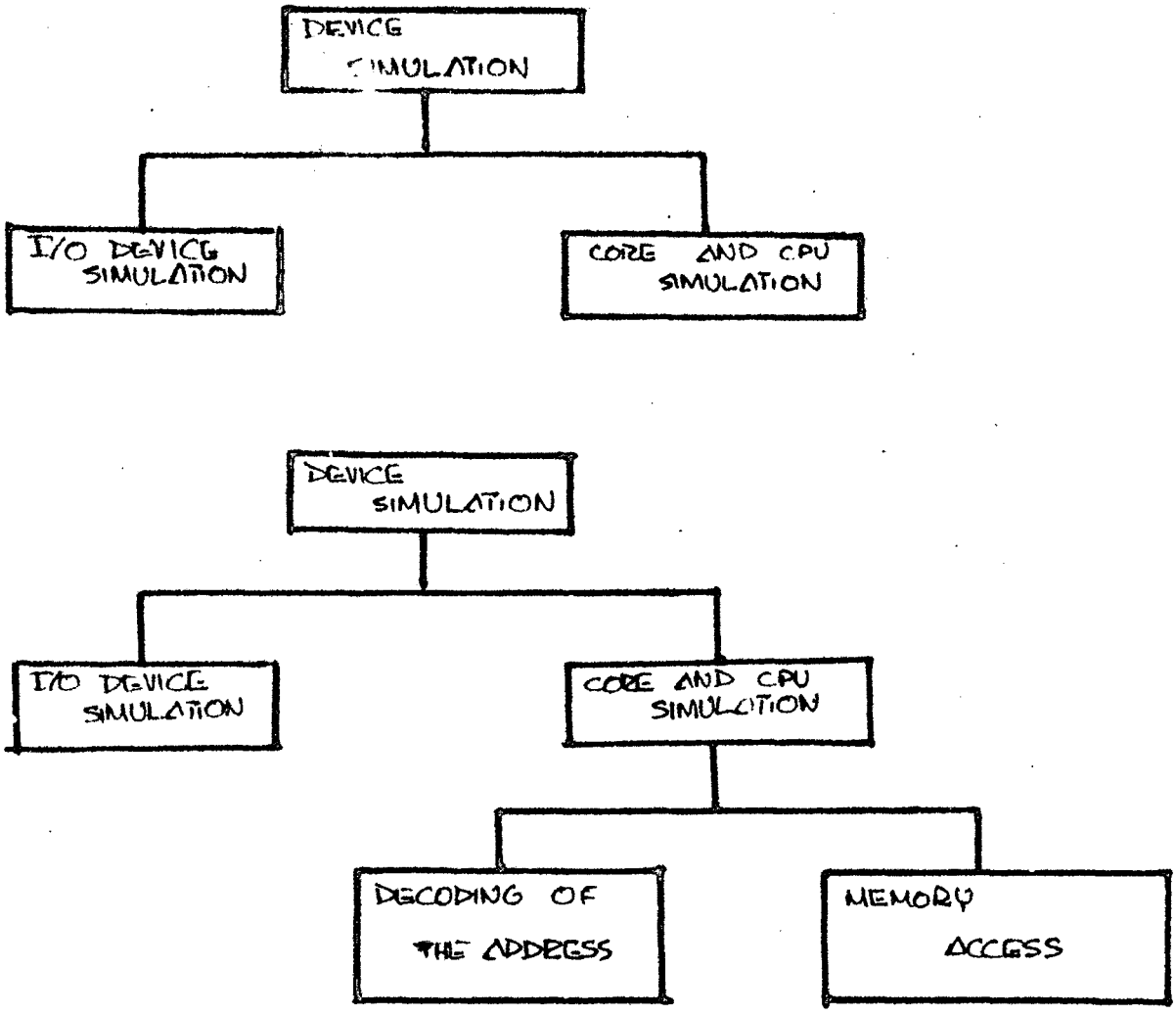
DEVICE
SIMULATION
I/O DEVICE
SIMULATION
CORE AND CPU
SIMULATION
DEVICE
SIMULATION
I/O DEVICE
SIMULATION
CORE AND CPU
SIMULATION
DECODING OF
THE ADDRESS
MEMORY
ACCESS

FIG. 3.

are pushed into another module. This second module is further divided into two parts one which decodes the given address and identifies the appropriate area to which the address belongs to. It may be noted that this routine is simulated machine dependent because this involves the particular configurration and absolute address of the area of the subsystem in the uni address space (REALTASKS).

The other module contains routines which initiate appropriate storage or retrieve routines (MEMORY). The reason for this modularisation is that the third module (MEMORY) does not involve the details of any particular system. These routines can well work for all 16 bit word and 8 bit byte oriented systems. Modules UNIBUS and REALTASKS contain the routines which are heavily dependent on the TDC 316 system. But these two are pushed into two different modules to enhance understandability and smoothen the interaction between the modules which use these modules. It is appropriate to keep all those modules which create and maintain I/O events and interrupts;and those that maintain the I/O block in the memory in one block. There fore, now we have the following structure of simulation task(See Fig.4.).

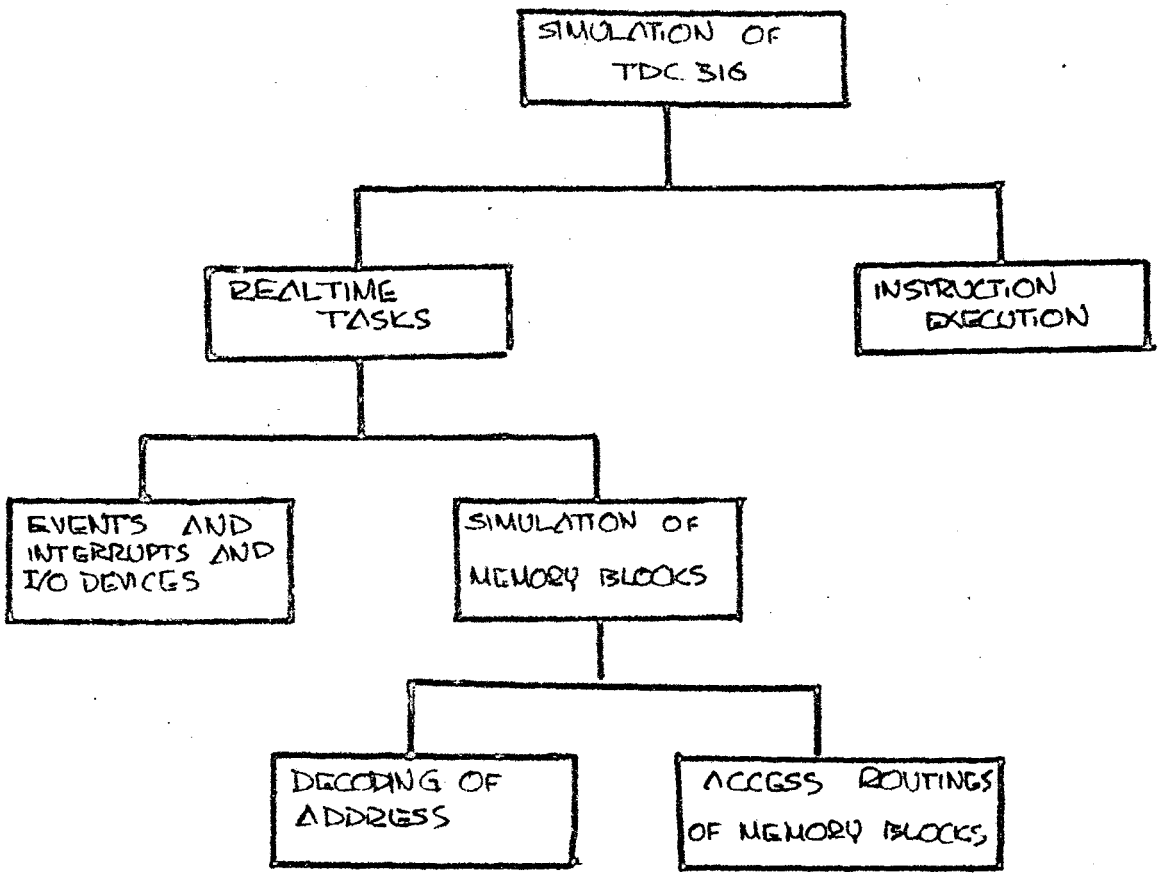
SIMULATION OF TDC 316
REALTIME TASKS
INSTRUCTION EXECUTION
EVENTS AND INTERRUPTS AND I/O DEVICES
SIMULATION OF MEMORY BLOCKS
DECODING OF ADDRESS
ACCESS ROUTINGS OF MEMORY BLOCKS

FIG. 4.

INSTRUCTION EXECUTION

Let us now discuss the simulation of instruction execution.

The process of instruction execution, in most computer systems, can be denoted by the succession of major states of their central processors. For a typical instruction the time succession of a typical c.p.u's major states would be:

(1) the instruction fetch state.
(2) operation decode.
(3) operand address evaluation.
(4) execute state; and finally
(5) conditional entry into TRAP state to check abnormal conditions internal to cpu.

Algorithm for these tasks is as shown in ALG.4.

This instruction execution part, we further divided into two groups: one that contains routines for instruction fetching, decoding and evaluation of source and destination address; the other part that does the actual execution of instruction.

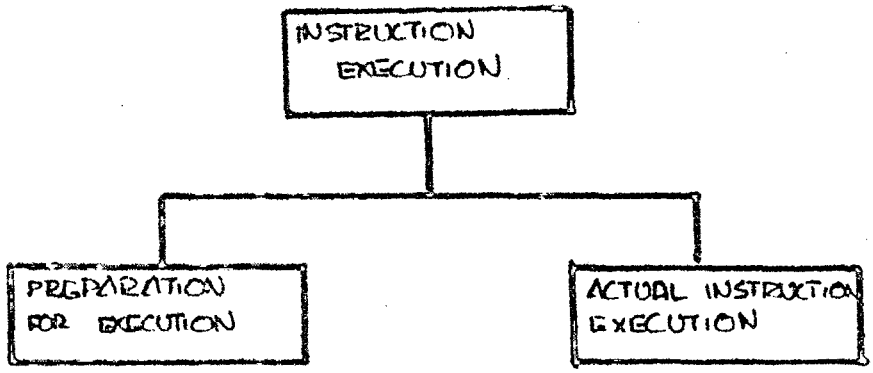

FIG. 5.

```
begin EXECUTION
  while INSTRUCTIONS ARE NOT EXHAUSTED do
  begin
     INITIALIZE EXECUTION DATA;
     CHECK FOR BREAK POINT TRAPS;
     FETCH INSTRUCTION FROM MEMORY;
     DECODE INSTRUCTION;
     EVALUATE SOURCE AND DESTINATION OPERANDS;
     EXECUTE THE INSTRUCTION;
     RECORD IF ANY BREAK POINT REQUESTS
        ARE PENDING;
     SERVE INTERRUPTS AND TRAPS
  end
end EXECUTION;
```

ALG4.

See Fig.5.

See ALG.5.

Fetch State:

The fetch state causes an access, by means of a next instruction pointer to the address space for fetching the next instruction. The retrieval process is successful if the pointer has an allowed value and there exists a definition for particular member of the address space accessed. Successful fetching of the instruction is followed by actions to update the pointer, preserve a brief history of pointers to provide a trail of the progress of a simulated program and to parse instruction word into various fields relavant to instruction formats. Failure of the retrieval process causes TRAP FLAGS to be turned on. (The trap flags for signalling disallowed pointer values are turned on by cpu routine for fetch process, and the trap flags for signalling non existent devices in the address space is turned on by the appropriate storage and retrieve routines).

See ALG.6.

Decode State:

This state is concerned with the identification of the instruction class (and hence the instruction format)

```
begin PREPARATION
   FETCH INSTRUCTION FROM MEMORY ;
   INCREMENT PROGRAM COUNTER ;
   DECODE INSTRUCTION ;
   EVALUATE SOURCE AND DESTINATION OPERANDS
end PREPARATION;
```

ALG.5

```
begin FETCH
   GET INSTRUCTION WHOSE ADDRESS IS IN PC ;
   INCREMENT PC ;
   INCREMENT 'REALTIME' BY 'FETCHTIME' ;
   If TRAP
    Then EXIT WITH ERROR MESSAGE
end FETCH;
```

ALG.6.

```
begin DECODE
   EXAMINE THE INSTRUCTION ;
   If (INSTRUCTION = RESERVED INSTRUCTION)
     Then SET APPROPRIATE TRAP FLAGS
     else OUTPUT THE OPCODE
end DECODE ;
```

ALG.7

to which a given instruction belongs. This procedure takes the instruction as input and gives the OPCODE of the instruction as output. This is typically dependent on the particular machine. This procedure also sets certain trap flags depending upon the opcode type (e.g. if the instruction is a reserved instruction). This preserves the purpose of the two modes of operations: user mode and kernal mode of TDC 316 central processor.

See ALG. 7 .

Evaluation of source and destination addresses:

The operation in the source state are, firstly, evaluation of the source operand address and secondly, the retrieval of the value of the source operand from the address space. In the destination state, the destinaton address is evaluated. The destination operand, if required at all, is fetched and used only in the execute state.

The address evaluation procedure of the TDC 316 is more elaborate than in most contemporary machines. All addressing is done via general purpose registers. The mode

```
function ADDRESS (MODE, REGISTERNUMB): return integer ;
begin
   EXAMING  REGISTER NUMB AND MODE ;
   CALCULATE  ADDRESS
end ADDRESS ;
begin OPERANDS
   FIND  MODE AND REGISTER NUMB OF
         SOURCE AND DESTINATION OPERANDS ;
   SOURCE := ADDRESS (SOURCEMODE, SOURCE REGNUMB) ;
   DESTINATION := ADDRESS(DESTMODE , DESTREGNUMB)
end OPERANDS ;
```

ALG. 8.

```
begin EXECUTE
   INITIALIZE  CONDITION CODES ;
   EXECUTE  INSTRUCTION ;
   UPDATE  PSW  AND  CONDITION CODES
end EXECUTE ;
```

ALG. 9

of usage of these registers in the address evaluation is specified by a separate field.

We have written a procedure to accomplish this. This procedure takes the instruction register and finds out the MODE and REGISTER of source and destination operands. Then calls another procedure which takes these two as input parameters and returns absolute address of the operand.

See ALG. 8 .

EXECUTE STATE:

This state forms the core of the machine language instruction execution process as it is here that detailed actions corresponding to each of the instructions in the machine language are performed. For execution of each the instructions, corresponding operations are performed and the condition codes are set as required by the outcome of the operation.

See ALG. 9 .

The simulation process is, now, as shown in Fig. 6.

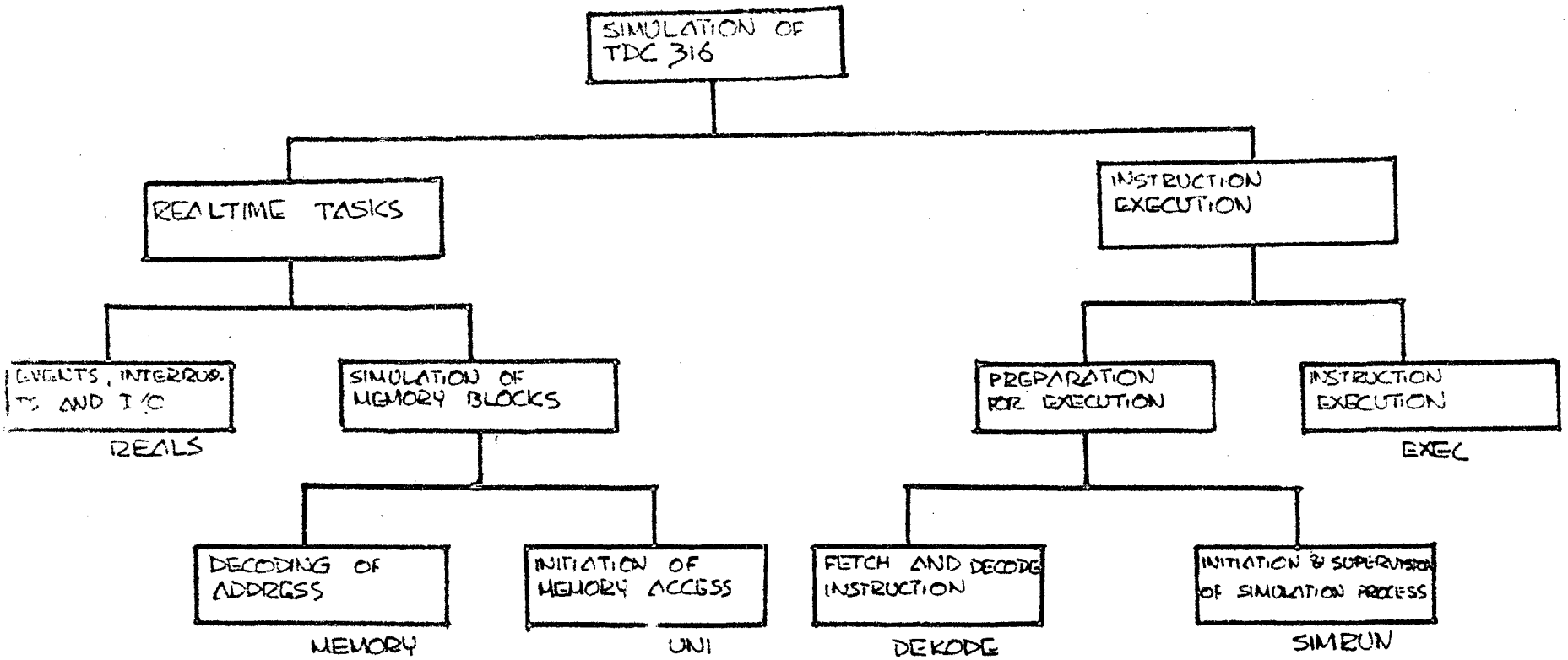
SIMULATION OF TDC 316
REALTIME TASKS
INSTRUCTION EXECUTION
EVENTS, INTERRUPTS AND I/O
SIMULATION OF MEMORY BLOCKS
PREPARATION FOR EXECUTION
INSTRUCTION EXECUTION
REALS
EXEC
DECODING OF ADDRESS
INITIATION OF MEMORY ACCESS
FETCH AND DECODE INSTRUCTION
INITIATION & SUPERVISION OF SIMULATION PROCESS
MEMORY
UNI
DEKODE
SIMRUN

FIG.6 .

CHAPTER II

MODULARIZATION METHODOLOGY

Dijkstra says programming is an art. However, much as artistic skills can be improved by suitable methods, programming skills can also be improved by adapting efficient methods.

It is recognized that the crucial part of programming is not just the runnable representation of the problem. It is the understandability, flexibility, portability, comprehensibility, reliability, and correctness, apart from runnable representation, make the crucial part of the programs.

The research in software engineering and programming methodology is mostly to achieve the above goals.

1. MODULARIZATION AND ITS ADVANTAGES:

The evolution of structured programming (Dijkstra, 13) and step wise refinement (Wirth, 18) methods are a few examples of researcher's success in this area. More primitive than these is the 'decomposition' of a big task into smaller ones and having an appropriate interface among the smaller tasks and the 'main' programs. The concept of procedures and functions (as they are refered to in the most of the higher level languages, typically Pascal) is

common in programming. This, in a very primitive sence, can be called as MODULARIZATION. A further development in this direction is the modularization as it is known today.

A module is a collection of declarations and is a unit for compilation which can reference, aside from its own variables, only those variables that are explicitly made available to it (V.R. Prasad, 15).

A large program is typically one that is developed and later modified by a changing group of programmers, who may be organized in number of programming teams. Inorder for such teams to work rather independently a program has to be structured into MODULES with clearly defined interfaces (J. Steensgaard and Madsen, 20).

Understandability of a program varies inversely with the amount of information one has to remember while going through the program text. The amount of such information becomes exhorbitant in the case of large monolithic programs. A good design would collect strongly related information into smaller modules and establish a fairly understandable interaction across such modules.

Dividing a big program into smaller modules in this fashion has come to known as MODULARIZATION (V.R. Prasad, 15).

In a modularized system any particular module needs to know only the relevant parts of the other modules and internal details of a module are invisible outside, unless explicitly designed to the countrary. This is known as 'need to know principle' (Parnas, 1972).

The benifits expected of modular programming are (Bergland, G.D., 8):

1. Managerial-development should be shortened because separate groups would work on each module with little need for communication.
2. Product Flexibility - it should be possible to make drastic changes to one module without need to change others.
3. Comprehensibility - it should be possible to study the system one module at a time. The whole system can therefore, be better designed because it is better understood.
4. Complexity - the control of the program complexity

> is the underlying, objective of most of the software engineering techniques. The concept of 'divic and conquer' is most important as an answer to the complexity, provided it is done correctly. When a program can be reduced into 'independent' parts, the complexity is reduced dramatically.

Consider the program A in the fig.7 where we have access to only the input and output. A noble goal would be to completely test this program by executing each unique path. In the example shown, there are approximately 250 billion unique paths through this module. If it takes one milli second to perform one test it would take about eight years to completely test all unique paths!! If however, we had knowledge of what was inside the program and recognized that it could be partitioned into two independent modules B and C, which have low 'connectivity' and 'coupling' (coupling is a measure of strength of interconnection between the modules), our testing job could be reduced. To test both of these modules separately requires that we only test about one million unique paths through each module which takes about 17 minutes !!

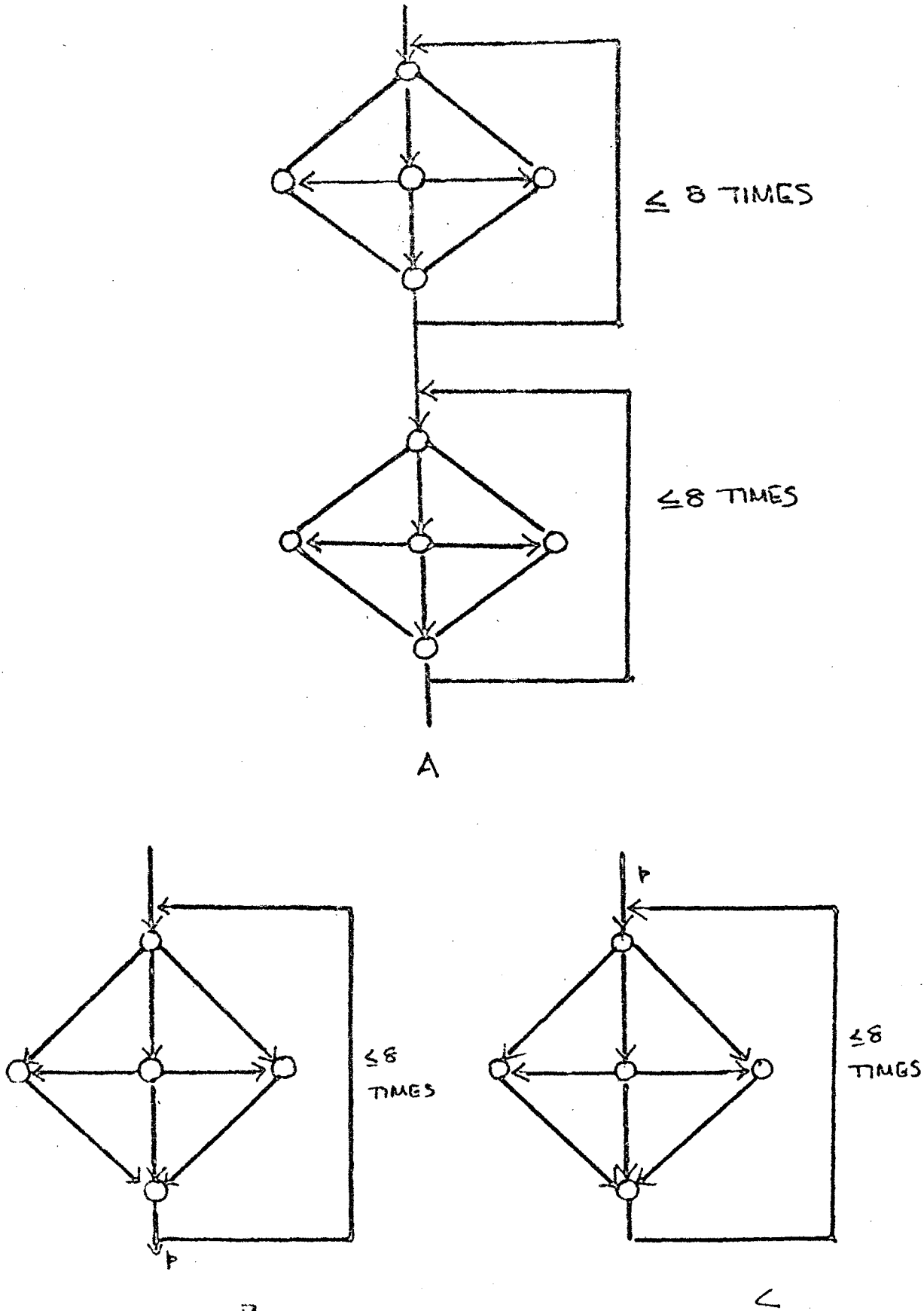

FIG. 7

Thus, it is understood that testing problem is best solved during design stage. It is impossible to exhaustively test any program of significant size.

5. Reliability-since each program can be developed compiled and tested separately, when they are combined to form a big system the finished product is expected to be of high reliability. In big monolithic programs the 'side effects' caused by various procedures and variables, types etc; can reduce the reliability of the product.

6. Understandability-as stated in the above paragraphs, the understandability varies inversely with the amount of information one has to remember while going through the program. Since separate modules are much smaller than the total program, understandability is greately enhanced.

7. Miscellaneous-scope rules in modular programming languages (example CCNPASCAL, EUCLID, MESA, ADA, CLU etc;) facilitate duplication of identifiers and procedure names. Hence, these names can be as suggestive as possible and can be duplicated as long as they are not 'exported' (CCNPASCAL notation) into any other module. Without this facility duplication of these names can cause severe confusions.

This advantage is distinct from the advantage one gets out of ordinary scope rules of the programming languages such as Pascal, in the sence that the modules can be developed by different persons.

All the above said benifits also result in another important advantage namely, the cost reduction in the development and maintenance of the system, which is of major importance to the commercial systems.

The other benefits which are subtle, nevertheless important, will be clear in our discussion about the structure of the modular programs and the modularization criteria. We may not list them out explicitly.

2. CONCEPTS OF MODULAR PROGRAMS

A. STRUCTURE OF MODULAR PROGRAMS

> 'Choosing the right program structure can lead to better programs' (Joshua turner, 9).

We shall now, discuss some useful concepts, analytically, regarding how the structure of the modular programs should be, in general and what is the structure of our simulator, in particular.

The goal of the structured programming is to produce programs that are at ones 'easy to understand', reliable in operation and are easy to maintain or extend. Structured coding of the individual modules can help achieve this goal, but ... the structuring of the whole program into modules can be a much more important factor: A program unstructured at module level (one in which modules can transfer control to other modules arbitrarily) is just as unmanageable as a single module in which GO TO statements are used indiscriminantly. Although a few contributions to the literature offer explicit distinction between good and bad program structures, a number of papers describe well-structured modular programs in general way. Wirth has described program structures in terms of levels of abstraction, where the lower level of structure represent ever increasing refinement of the higher levels (Wirth.m.). Dijkstra has described each level as constituting an abstract resource or virtual machine providing a specific capability to the level above it, and supported by levels below it (Dahl, Dijkstra, Hoare structured programming 1972 13).

Well-structured programs are most commonly characterized as 'hierarchical' or tree-structured'.

These terms are not often well defined. But a hierarchy is defined informally as any program structured in levels where the module on a given lower level may or may not be shared by the modules on the higher levels. The extent of sharing of the modules at the lower level is important: if too much sharing is allowed it becomes impossible to separate the functions performed by one branch of the hierarchy from the functions performed by other branches. As a specific type of hierarchy, 'tree structure' is defined informally as a hierarchy in which sharing is allowed. In a pure tree structure, a given lower level module may be called only by a single module on the next higher level. See Fig. 8.

It is believed that for a large class of programming problems the pure tree structure, with a few desirable exceptions, is the best model to choose in designing modular programs (especially systems programs), but it seems ideal for most data processing applications intended for general purpose computers. A module should have functional strength. A module has functional strength if all of its elements contribute to the performance of a single

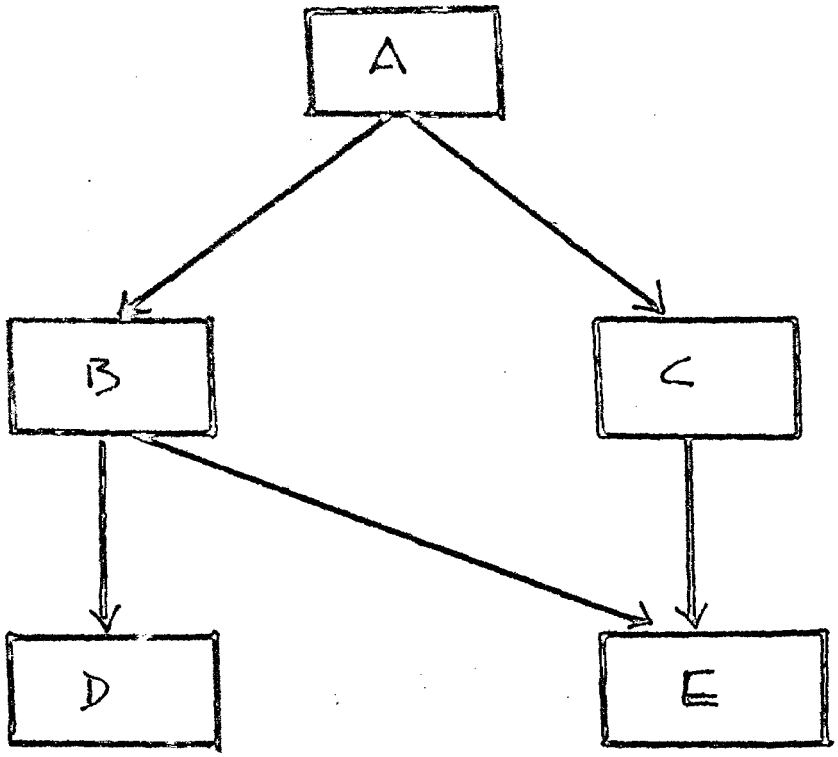

FIG. 8

specific task (Yourdon E, 11). A good test of whether a module performs a single task is whether the purpose of the module can be stated in one simple sentence. For example the module EXEC in our SIM16 is functionally strong. Module in a pure tree structure is initiated through a single 'main' or 'root' module. The root module is responsible for control over the entire collection of modules. The module DIALOG is 'main' root of the simulator.

Each of the modules directly called by the root module controls a major branch of subtree of the program. For instance the module SIMRUN is directly called by DIALOG and SIMRUN forms the root of another subtree which makes calls to DEKODE, EXEC, MEMORY. Each branch of subtree of such a tree structure is functionally and structurally separate from the other subtrees. The module DEKODE performs all the operations necessary to decode a single instruction and passes the values of opcode, and addresses of the operands.

A program having a pure tree structure has several fundamental characteristics; first, at each level, each module has exclusive control over the modules it calls.

And second, each branch of the program is isolated from the other branches of the same level. This is a fundamental consequence of the restriction against the sharing of lower level modules.

There is one final requirement necessary to ensure that each branch of the tree remains isolated from other branches at the same level. That requirement is called 'locality of reference'. The structure of the simulator preserves the 'locality of reference'. There is no interaction between modules FILEOP and SIMRUN; similarly no interaction between REALS and DECODE.

A program retains locality of reference whenever all communications between modules is by means of the parameter lists passed between a calling program and the module it calls. Such a program may be said to be 'parameter driven'. Locality of reference is lost whenever modules are allowed to communicate by any other means, such as through shared stozage area. When locality of reference is violated, it becomes possible for one branch of the tree to interact with another branch without the knowledge of the higher level modules which supposedly control the respective branches.

A picture of the design of a modular program as a tree structure can now be summarised as:

(1) it should consist of a collection of modules each of which has 'functional strength';
(2) there should be no sharing of modules between branches of the tree;
(3) and program should be parameter driven i.e.,; locality of reference should be preserved.

Advantages of the pure tree structure:

What are the advantages of such a program structure? The primary advantage of the tree structures has already been mentioned.

As a consequence of the restriction against the sharing of lower level modules and because of the requirement that locality of reference be preserved, the tree structure is able to maintain a complete separation of functions performed by each branch of the tree. Each branch performs its own functions, with minimal relationships to the functions of the other branches. The relationships among the elements not in the same branch are minimized. These advantages are consistent with goals of 'structured

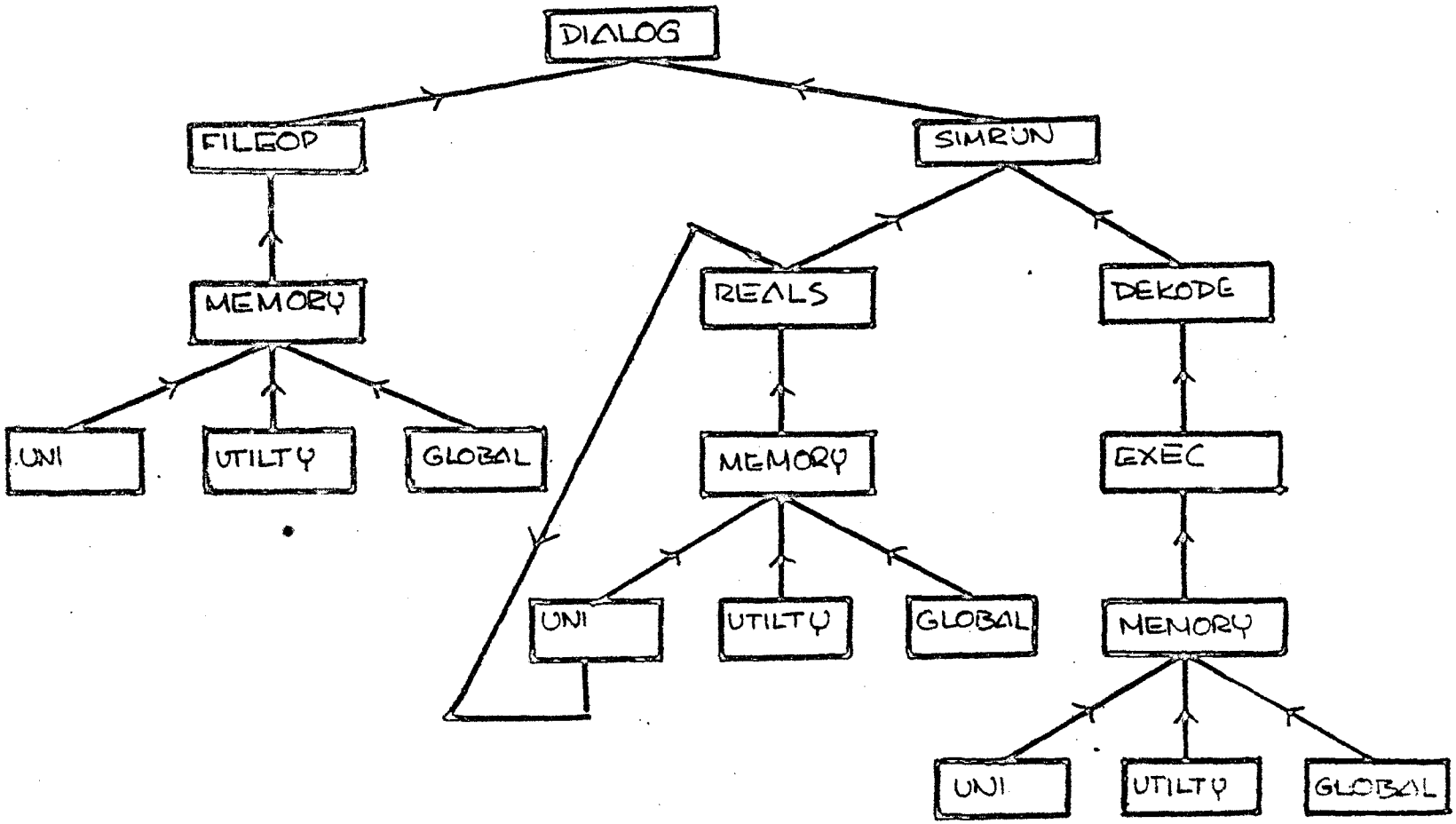
DIALOG
FILEOP
SIMRUN
MEMORY
REALS
DEKODE
UNI
UTILTY
GLOBAL
MEMORY
EXEC
UNI
UTILTY
GLOBAL
MEMORY
UNI
UTILTY
GLOBAL

FIG. 9

programming'. The program is 'easier to understand' because each level of the program is structured into a number of separate branches. Since each branch is conserved exclusively with its own functions within the program, the chances for unwanted and unforseen interactions between modules are minimized, resulting in a finished program of 'high reliability'. Finally because of the separation of the functions within the program, 'maintenance is easier', since maintenance can be performed on any one branch of the tree, with minimal impact on the other branches. A second advantage of the pure tree structure, closely relates with the separation of the functions within the program is that such a tree structure reduces the complexity of the program by minimizing the number of interactions between the modules. It is said that (Aron, 12) in the worst case, a network structure containing a modules in which each module interacts with every other module is N(N-1)/2, or roughly proportional to N square. In any hierarchical structure, however the number of interactions will be reduced. The number of interactions depends directly on the extent of which the lower level modules are shared. As the tree

structure becomes less 'latticelike' and more 'treelike', the number of interactions are minimized, i.e. each module is called by exactly by one higher level module, and therefore the number of interactions is proportional to N.

How important is it to minimize the number of interactions between the modules? It is noted (Yourdon,11) that we can expect the number of bugs in a system to grow as the number of interactions grows. Therefore, minimizing the interactions is of great importance.

In our simulator we tried to minimize the insteractions between the modules as far as possible, by introduction the module GLOBAL which contains some frequently and widely used data items and controls the access to them.

Here, we have many things to discuss about our simulator. Firstly, not all of the above said criteria are met in our simulator. In fact,in any large system not all of them can be,'efficiently',met simultaneously. There have to be some deviations from pure tree structure (Joshua, 9). In any

large system there are number of activities which are performed separately at many different points. Where a particular function is required in a number of places, the urge is almost irresistible to code a single module to perform that function and allow that module to be shared. The alternative would be to record that function wherever needed. Thus module sharing allows the program designer to reduce, the coding effort. It is for this reason we allowed the sharing of the modules MEMORY, GLOBAL, and UTILITY. MEMORY is the module which contains routines to access the simulated memory locations. Since at various places the memory access is needed (for example: for loading and debugging purposes in FILEOP, for trap and interrupt handling in REALS, to fetch and dekode the instructions in SIMRUN, for execution of instruction in EXEC). Therefore, this module must be shared by the other modules at higher levels. Similarly the module GLOBAL. In fact, the module GLOBAL is introduced to follow a particular discipline namely 'discouraging the loops or mutual interactions across the modules';that is, no module at lower level should 'use' a

module at higher level. The introduction of a new module such as GLOBAL is not a perfect remidy of the mutual interactions in view of the data abstraction criteria (parnas;1). But, still it is better than allowing mutual interactions or loops among the modules.

Another advantage of module sharing is that, it allows us to conceal superfluous details of the implementation of a particular function from higher level modules which make use of that function. If a given function were hard codedwhenever needed, then each module that required the function would have to know the details of its algorithm. The possibility would arise for inconsistencies in implementations, as well as for high overhead associated with maintenance to the function.

In our simulator structure we also allowed a 'loop' among the modules REALS, MEMORY, and UNI. The module REALS uses MEMORY to access memory locations to serve the interrupts and traps. As explained in the Chapter 1 modules MEMORY and UNI do the same function of accessing the memory locations but UNI contains the machine dependent features of the machine being simulated such as deciding about to

which memory block the given address belongs and which special registers are to be accessed and so on. Therefore UNI is used by MEMORY. The routines which access I/O area memory block are contained in the module REALS. This is done because of the data abstraction criterion which is one of major criteria in decomposing the system into modules (Parnas, 1).

It is better understood if I/O area memory access routines are kept in the module which simulates the real time tasks because I/O device maintenance is afterall a real time task, they can enter event or interrupts depending on the conditions. So, we had to choose a compromization between data abstraction and mutual interaction. We have preferred data abstraction because of the two reasons:

(1) This 'loop' occurs at the lower most subtree and its 'parent' module. If it had been a module and its far off 'ancestor' we would not have done this. (note: MEMORY and UNI can be for all practical purposes viewed as a single module).
(2) The crucial and confusing part of the simulation

task is the real time simulation. Therefore we felt that it is necessary to increase the understandability of the modules which perform the real time task simulation.

Thus, it is demonstrated through our simulator that pure tree structure is not always an ideal one for modular programs though the pure tree structure has many advantages over the alternate ones. While designing the structure of modular programs one should never 'force' a tree structure at the cost of the other criteria such as data abstraction and understandability and so on. If the tree structure 'naturally' evolved out of the system logic it is always preferable. Otherwise one has to choose a proper compromisation.

B. CRITERIA OF DECOMPOSING SYSTEMS INTO MODULES

So far we have been discussing the structure of modular programs. We have discussed what is an ideal structure and the advantages (why) in such an ideal structure and the desirable exceptions to using that ideal structure. We have also discussed simultaneously how far our

simulator adhered to the above said structure and the reasons for the deviations from that ideal structure.

We shall now discuss the criteria to be used in decomposing the systems into modules and how they are met in our simulator.

The effectiveness of a modilarization depends upon the criteria used to decomposing the system into modules (Parnas, 1). One of the criteria generally used is to make each major step in the processing a module. According to this criterion we should have had, in our simulator, preparation for execution and instruction execution in the same module. But this is not a prime criterion. The prime criterion would be the 'information hiding' (Parnas, 1).

According to this criterion modules no longer correspond to steps in the processing. The instruction execution module EXEC, for example, is separated from the module for preparation for instruction execution, SIMRUN. Also, the machine details of decoding an instruction, evaluation of opcode and instruction address of operands are 'hidden' from SIMRUN and kept in the module DEKODE. DEKODE is used

only by SIMRUN, thus, hiding the detailed information from the user of SIMRUN.

Another example is that the simulated machine details of memory access such as decoding the address, reading from or writing into the specified memory blocks of special purpose registers are separated from MEMORY module and put in the module UNI. UNI is used only by MEMORY, thus hiding the machine details from the user of the module MEMORY. Thus, the 'information hiding' criterion is given much importance in decomposing the simulator modules. The reason for doing so is to achieve our goal of developing a simulator which can be easily modified to implement for a different configuration of the system or for another 16 bit machine.

If the simulator is to be implemented for another configurration of the system where the instruction formats are changed or memory block addresses are changed then only the module DEKODE or UNI respectively need to be changed. Similarly, if only the instruction set differs from the existing machine then the module EXEC only needs modification without the concern of the other modules.

In addition to the general criterion that each module hides some design decisions or implementation details, we can mention one specific examples of decompositions which seem advisible.

A data structure, its internal linkages, accessing procedures and modifying procedures are part of single module. They are not shared by many modules as is conventionally done. For instance, the general purpose registers of TDC16 are declared in the module UNI so is the procedure to modify these contents REGSET which takes as parameters the register number and the newvalue. Similarly, the 'trapflags' which are declared in the module GLOBAL are allowed to be modified from any other module only by the procedure SETTRAP declared in the GLOBAL itself. Finally, we shall give some concluding remarks regarding the structure of the modular programs and the criteria to be used in decomposing systems into modules.

In the discussion of system structure it is easy to confuse the benefits of a good decomposition with those of a hierarchical (or tree) structure. But it is argumented that hierarchical structure and clean decomposition are two

desirable but 'independent' properties of a system structure. We also have demonstrated this point in our simulator. Sometimes they may become contradictory to each other in which case one has to make a suitable choice as we have done in our simulator (by sacrifying tree structure for information hiding and understandability). It is also important to design the system structure which is best suited to the logic of the program and the aims of the program development. For example our aim is to separate the simulated machine dependent parts from the independent parts. Therefore, we have separated the modules DIALOG, SIMRUN, MEMORY, UTILTY which are 'almost' configurration independent from the other modules such as FILEOP, DEKODE, UNI, and EXEC respectively.

We have also said informally as a part of our discussion how this simulator can be modified for other configurrations of the same system or a class of 16 bit machines.

In fact, any one who has ever attempted the design of large system realizes the importance of having a well-defined structure - any structure - as a tool for organizing

thinking. As Dijkstra and Hoare pointed out, the scope of the computations we are attempting to program is already well beyond the grasp of our unaided minds. So 'we must organize the computations in such a way that our limited powers are sufficient to guarantee that the computation will establish the desired effect' (Dahl, Dijkstra, Hoare, 13).

CHAPTER III

DESIGN & DEVELOPMENT OF THE MODULAR SIMULATOR, SIM16

The simulator SIM16 is written in CCNPASCAL language, a Pascal kind of language, with additional features to support:

Modular and concurrent programming

Type parameterization and

Data abstraction facilities. The simulator is developed on DEC system 1077 At NCSDCT-BOMBAY. Simulation is achieved by program code for host machine imitating the behaviour of TDC 316 configuration. Sequential simulation of paralel process (CPU and devices) is done by time sharing.

The simulation configuration is:

One central processor

One key board/teleprinter

One high speed paper tape reader/punch

One 20 mili second line frequency clock.

Some special features are:

Effective console simulation, interpretive ODT

Easy handling of simulated I/O devices.

We have developed the simulator program in a stepwise refinement method. The first two steps of the refinement are listed in ALG.1.

Initially the simulator was developed on CDC6600 in BCPL language by V.R. PRASAD (PRASAD, 17). Lateron, this was modified into FORTRAN CUM MACRO10 on DEC system 1077 by V.R. PRASAD. The original BCPL version is contained in five files with very strong interaction across the various files. The FORTRAN CUM MACRO10 version (50% of MACRO10 - The assembly language of DEC system 10) is a big monolithic program.

This BCPL version is taken as the base and developed further. First, the program is translated into Pascal again in five different files. Here the distribution of information into various relevant files becomes important. The five modules are the following:

DIALOG PAS: This file initiates dialogue with the terminal user and processes the user commands. This also provides on line debugging facility. This file makes calls to different routines declared in the other files. Thus, this is a sort of controlling file of the whole program.

SIM.PAS: This file simulates the major instruction cycle of the CPU. The five parallel process clock, KBD, EVENT, INTERRUPT, AND CPU are simulated by interleaving in time. On occurrence of special commands such as halt, single instruction mode set, address break, I/O device not ready end-of-file on ASCII file; control is returned to DIALOG to accept the user commands.

UNIBUS.PAS: This file declares the data structures required for unibus simulation and defines various access rules. The data structures include 56K bytes of TDC16 core memory (14K words of DEC system 10 memory), cpu registers and relavant device registers. This file also defines some miscellaneous routines for various operations on memory word/byte.

RUN.PAS: This file is used by SIM for the interpretation of a single instruction. This file contains the routines to find the effective address of the operands taking the mode and register number as inputs; to decode the supplied TDC316 instruction and producing the mapped opcode as value. And finally, a routing to fetch and decode the instruction, to find the source and destination operands.

EXCUTE.PAS: This file defines a single routine which does the actual execution of the decoded instruction. All the information required by this routine is supplied through global variables.

PROGRAM DEVELOPMENT

The program is, then, converted into CCNPASCAL language. At this point major shuffling of the information in various modules is done to suit to

(1) the aims of the design
(2) ideal structure (hierarchical) of the modular programs and
(3) the standard criteria to decompose the system into modules.

Firstly to develop the CCNPASCAL modular version we fixed the 'context' of each module (see appendix 1 for CCNPASCAL description). In fact, from our experience we also advocate the programmers to fix up the context of the each module first before coding any large system. This can be done only after having a clear idea about the over all system design. This is the most crucial part of the whole system design, for, this is the stage at which the

structure of the program is defined and fixed. The designing aspects of the simulator program are described below with reference to the five pascal files described above.

DIALOG-PAS: This file is split up into two modules: DIALOG and FILEOP.

The module DIALOG now, contains only the procedures to interact with the terminal user, to initialize the system, to initiate loading of the test program, to initiate on-line interpretive debugging, and start with the simulation process and finally to provide the summary of the simulation process.

The module FILEOP contains the routines to operate on the files, such as loading the test file and dumping the binary file into a user specified file.

The reason for above splitting is to hide the details of file operations which are dependent on the configuration of both TDC16 and DEC system10 from the user of the module DIALOG. This makes DIALOG configuration independent and thus, paving our way in the direction of fulfilling our objective.

SIM.PAS: This is one of the modules which is shuffled thoroughly. The routines which perform the realtime tasks such as entering into an event and interrupt, serving them, doing trap actions and simulating I/O devices, are now put in the module called REALS.

This module REALS also contains the routines to read and write from I/O area memory block. These routines were previously residing in UNIBUS.PAS file.

The reason for this distribution is the data abstraction criterion. All the information which is concerned with the real time tasks is put in one module.

The main program in the original file SIM.PAS which carried over the main process of simulation (which resumbles the algorithm 1 version 2 given above), along with the main program of the original file RUN.PAS which is responsible for the interpretation of a single instruction, are clubbed into one module called SIMRUN.

The reason is primarily information hiding criterion. The details of the real time process is separated from the user who initiates those real time

process. This module is also ,to a good extent,configuration independent, because the details of decoding an instruction to find opcode and operand address are no longer visible to the user of SIMRUN. (they are kept in the module DEKODE), which is the key module for simulation.

RUN.PAS: This, as we have discussed just in the above paragraph, loses its identity by getting distributed into the modules SIMRUN and DEKODE. The module DEKODE consists of the routines which decode the instruction to give opcode and the routines to find the address of the operands.

UNIBUS.PAS: This module, now, split up into two modules: MEMORY and UNI. UNI is the module which actually contains the routines to decode the given address to find out to which block it refers and to do the reading and writing operations on words and bytes. This is strongly dependent on the simulated machine TDC316. The other module MEMORY contains routines to initiate the reading and writing operations on words/bytes of memory blocks such as simulated core,CPU and I/O devices. This contains no details of the simulated machine structure.

The reason is all the other modules which access the read write operations will access only MEMORY. The machine structure details are 'hidden' in the module UNI. The module UNI is used only by MEMORY.

Thus, this splitting is done for two important reasons:

(1) The information hiding criterion and
(2) Most relavant to us is, separating machine dependent and the independent parts.

There is a 'loop' among the modules REALS, UNI and MEMORY. The reasons for allowing this 'loop' are discussed in the previous chapters.

EXCUTE.PAS: This is the module which almost retained its original form. No major modifications are made except that the access to the variables decalared outside this module is done through procedures declared in respective modules.

Apart from these modules we have two more modules GLOBAL and UTILTY.

GLOBAL: This module contains some widely used data items and procedures to modify these data items. The module GLOBAL is introduced to reduce the 'strength of interaction' between various modules.

This technique may not be in accordance with the data abstraction criterion that the related items should be grouped together. Nevertheless , this is introduced to reduce the mutual interaction among the various modules. Without the introduction of this module it would have been quite problamitic for having a 'clean structure' for the program.

The module UTILTY contains miscellaneous routines to perform various operations on memory words which are not generally available as library routines in higher level languages. Some examples are: accessing any portion of a given word including a single bit;and for logical AND ing/OR ing operations of two words; logical shifting and extending an 8 bit byte or 16 bit word to form a full 36 bit (DEC system 10) word and so on. This module is purely configuration independent and can be used for all 16 bit word, 8 bit byte oriented computer systems. This module

along with the modules MEMORY and GLOBAL is used throughout the program by all modules.

Thus, sharing of the lower level modules by the higher level ones is allowed in our program structure for obvious reasons.

This finishes the description of the design and development of the simulator program.

PART B:

PROBLEMS ENCOUNTERED IN THE DESIGN AND DEVELOPMENT:

Let us now discuss the problems we have encountered in the design and development process of the simulator. We also discuss how we have overcome the problems encountered.

The first problem that is encountered is right at the design stage, in fixing up the 'context' of each module. The problem is that what data items are to be kept in which module. Making a decision about this was critical. In general there are many criteria for decomposing the systems into modules (Parnas D.L.). But the problem is which criterion is to be more important than the others.

We have analysed the information very carefully and decided which are the data items that are related to the

information to be contained in that module. At times it was 'tug of war' between two modules (may be, three modules). For instance, the variables SRC (source operand) DSTADD (destination address) are 'equally' relavant to the modules DEKODE, SIMRUN and EXEC.

The diciplene we have followed to make decision about the distribution of information is that if module A uses the module B then the data items which are required by both the modules are kept in the lower level module, that is module B. With that choice the above variables are kept in EXEC.

The other problem is also at the design stage. In fact, this problem was much more severe than the above problem.

There are certain data items such as TRAPFLAG s: MODE s of the device operations REGMODE, SPMODE (STACK pointer mode), HSRMODE and HSPMODE; REALTIME (which is used to keep track of the actual execution time on the simulated machine TDC316); the boolean variables which keep track of the running of the simulation process such as RUNMODE, HALTFLAG, WAIT: which are very widely used throughout the

program. These variables were causing mutual interaction across various modules.

We have overcome this problem by identifying such data items widely used, from the whole program, and grouping them together in one module called GLOBAL. This module is shared by all the other modules (except FILEOP) and the problem is solved !!

But for these problems, the design of the system was fairly smooth.

Another aspect worth mentioning here is the following. We have fixed the 'context' of all the modules first, and then filled in the code. The reason is that when there is any mutual interaction between two modules problem is which module is to be built first? This problem is well-solved by fixing up the context of both the modules, that is, the items that are exported out from each module. Then, when the code body is filled in, there will not be undefined variables existing in both the modules.

SECTION 2.

As we have been mentioning throught, our aim is to

develop a 'flexible' simulator which can be implemented for a number of systems/configurations 'easily' with little modifications. That was one of our main criteria in designing the structure and decompositing the system into modules.

We shall describe now how our simulator can be implemented for some other configurations of TDC 316 or a class of 16 bit machines.

We emphasize one point that this simulator can be used to simulate TDC 316 like machines, the systems having 16 bit word length and 8 bit byte addressible systems. Any other type which is drastically different from TDC 316 system may be difficult to be implemented. There are a number mini and micro computers which are commonly used. For instance, PDP 11 series, MOTOROLA 6800, Z 80, INTEL 8085 TEXAS TMS 9900, HP 2100 are some systems which are popular 16 bit machines.

Firstly the system/configuration independent modules are identified to be DIALOG, UTILTY, MEMORY, to a great extent SIMRUN. The machine dependent modules are FILEOP, REALS, DEKODE, UNI and EXEC.

Normally, for 16 bit systems the instruction length are 8(for byte instruction) and 16(for word instructions). Therefore, the module FILEOP does not need much modification.

The module REALS needs major changes especially for the routines IODONE, IOREAD, and IORITE. Since the module DEKODE involves the instruction formats of the particular system, this module needs to be changed, greatly or perhaps complete replacement is needed.

Similarly the module UNI involves absolute address of different memory blocks. This needs major changes or even complete replacement.

In the module EXEC, the major function is the execution of the instructions. Though this module is dependent on the system heavily, most of the instruction are common to many systems. In development of this module we have used mostly the routines that declared in other modules. The advantage is that not too many machine dependent features are involved in executing most of the instructions.

But, the instructions that are system dependent are to be changed. Apart from this, the enumeration type

TYPEOPCODE also needs to be changed.

There is one point which the person who replaces the existing module is to be careful about. The exported declarations should be treated with great care in modifications or replacements or else inconsistencies in types of the declarations or the parameters in procedures or functions might cropup.

Best thing is to study the module(s) which uses the module being modified and then make the modifications.

Since there is smooth interaction across various modules and understandability is one of the main criteria of modularization, the logic of the program can be well understood, and hence not much difficulty in making modifications to the modules is expected.

SUMMARY AND CONCLUSIONS

We shall now,briefly summarize what has been said in the above chapters.

Objective of the dissertation is to carryout an experiment using modular programming with a particular goal of developing an easily modifiable software system. As an instance, we have taken the task of modularizing a semulator for computer system TDC316 which was already developed as a big single monolithic program. Modularization of this program is carried out to make it more 'flexible' i.e., to facilitate its implementation for a number of different configurations/systems.

The design strategy adopted in modularizing the simulator is to separate the simulated machine dependent and independent parts of the program.

The structure of the modular simulator is maintained almost hierarchical; 'information hiding' and understandabilit criteria are given preference in the decomposition of the system into modules. This makes the simulator 'flexible' by allowing easy modifications of the modules more or less independent of the other modules. The simulator now,is a sort of 'general system' in the sence that the repetition

Of certain tasks of simulation, when implementing the same program for other computer systems/configurations, is eliminated by the separation of simulated machine, dependent and independent parts. Therefore, to simulate a different system/configuration, only the machine dependent modules need to be modified/replaced.

The simulator is developed into ten CCNPASCAL modules. They are:

1. DIALOG
2. FILECP
3. DEKODE
4. SIMRUN
5. REALS
6. MEMORY
7. UNI
8. GLOBAL
9. UTILTY
10. EXEC

We have identified the system/configuration independent modules as DIALOG, UTILTY, MEMORY and to a good extend SIMRUN. Some of the remaining also need to

be changed only partially to implement the program to a new configuration.

One of the important problems we have encountered was at the design stage. There are certain data items which are widely used throughout the simulator program. These items were causing 'strong' interaction across different modules. This problem was overcome by grouping such widely used data in one module and allowing it to be shared by the other modules.

We suggest the programmers to follow the modularization methodology in developing large systems in view of its tremendous advantages. The structure should be as close to 'treelike' structure as possible. But one should never 'force' a tree structure to the program by sacrifying many other important criteria in decomposing system into modules.

There may not be universal criteria in such decomposition, but among the main criteria (discussed earlier), one should select appropriate ones which suit to the aims and goals of their project. Some times it may necessary to form one's own criteria for decomposition depending on the needs and necessities.

Once again we emphasize the point that our simulator can work well for only 16 bit computer systems. Any system which is drastically different from TDC 316 system necessitates drastic changes in the SIM16. In any case post developmental modifications to this simulator can be easily carried out.

We finally end this dissertation by quoting E.W. Dijkstra:

'In computer programming our basic building block has an associated time grain of less than a micro second, but our programs may take hours of computer time. I do not know of any other technology covering a ratio of 10 power 10 or more: the computer by virtue of its fantastic speeds, seems to be first to provide us with an environment where highly hierarchical artifacts are both possible and necessary. This challenge, viz, the confrontation with the programming task is so unique that this novel experience can teach us a lot about 'ourselves'. It should deepen our understanding of the process of design and creation; it should give us control over the task of organizing our thoughts. If it did not do so, to my taste we should not deserve the computers at all !!

It has already tought us a few lessons, and the one I have chosen to stress (in this talk) is the following. We shall do much better programming job, provided that we approach the task with a full appreciation of its tremendous difficulty, provided that we stick to modest and elegent programming languages, provided that we respect the intrinsic limitations of the human mind and approach the task as VERY HUMBLE PROGRAMMERS.'

APPENDIX 1

MODULAR FEATURES OF THE CCNPASCAL

CCNPASCAL has been designed to support ease in program development, understandability and modifiability. We will present here a brief note on the modular features of the language CCNPASCAL. The syntax of the language is more or less similar to that of Pascal. However, for the details of the CCNPASCAL syntax and semantics technical report on CCNPASCAL (V.R. Prasad, 15) may be referred to.

The 'unit' of compilation in CCNPASCAL is either a module or a program. A software system may contain several separately compiled units which, when intefaced using contexts, behave as if they were all compiled together.

A.1 MODULE STRUCTURE

A module is a collection of declarations and is a unit for compilation.

Syntax

⟨module⟩ ::= module ⟨module-id⟩ [⟨unit-import⟩]
[⟨unit-include⟩] ";" [⟨decl-seq⟩] end
⟨module-id⟩ "."

⟨module-id⟩ ::= ⟨identifier⟩

⟨decl-seq⟩ ::= {";" ⟨declaration⟩} [";"]

⟨declaration⟩ ::= ⟨context-decl⟩ | ⟨const-decl⟩ |
⟨type-decl⟩ | ⟨var-decl⟩ |
⟨routine-decl⟩

Example

```
module IODRIVER import DEVCXT = 'DSKC: DEVICE.CXT';
  type export BUFTYPE = array [1..128] of integer;
  procedure export DOIO(CHANNEL:cap; var BUF:BUFTYPE);
    begin...end DOIO;
  function export STATUS(CHANNEL:cap) return boolean;
    begin...end STATUS;
end IODRIVER.
```

Semantics

A module must be given a name (denoted by ⟨module-id⟩) that may be used by the implementation. A module definition may specify a unit-import clause (explained in section A.3) to be able to use precompiled units. A module may specify

a unit-include clause (also explained in section A.3) to effect textual inclusion of other units before it is compiled.

A.2 CONTEXTS

The designer of a module may want to provide different views of a module to different users. A context represents a particular view to a module by abstracting the specifications of the items that provide the view. For convenience, every module is associated with an implicitly declared context, called the 'implicit-context'. The implicit-context is identified by the module-id and collects together the specifications of all 'export declarations' (An 'export declaration' is one of a const, type, var, procedure of function declaration in which the identifier being declared is prefixed by the keyword export) appearing in the outermost level of the module definition. Compilation of a module also generates a context corresponding to each explicit context declaration (see section A.5) appearing in the module.

Thus far, the term 'specification' has been used in an informal sense. In general, the specification of a module

consists of all the information necessary to use the module correctly. It is upto the implementer to decide what a specification should actually consist of.

For example, the implicit-context corresponding to the module IODRIVER of the previous section may be encoded (in a particular implementation) as follows:

```
context IODRIVER;
  type BUFTYPE = array [1..128] of integer;
  procedure DOIO(CHANNEL:cap; var BUF:BUFTYPE);
  function STATUS(CHANNEL:cap) return boolean;
end IODRIVER.
```

Implementation of contexts is not defined by the language. For implementations supported by a filing system, a context may be implemented as a file . This has the advantage that protection on contexts can be implemented as protection on files. To simplify understanding it is assumed that a context is implemented as a file and follwed the convention that the implicit-context corresponding to a module M is represented as the file 'M.CXT'.

A.3 UNIT IMPORT and INCLUDE

A unit-import clause specifies a list of contexts

that are imported into a module before it is compiled. A unit-include clause lists names of modules (or programs) that must be textually included in a module before it is compiled.

Syntax

⟨unit-import⟩ ::= import {";" ⟨import-context⟩}

⟨unit-include⟩::= include {"," ⟨source-unit⟩}

⟨import-context⟩ ::= ⟨context-id⟩ ["=" ⟨string⟩]

⟨source-unit⟩ ::= ⟨unit-id⟩ ["=" ⟨string⟩]

⟨unit-id⟩::= ⟨module-id⟩ ¦ ⟨program-id⟩

Examples

```
import MYCXT
include ARITHPACK

import DEVCXT='DSKC:DEVICE.CXT'
include ARITHPACK='SYS:ARITH.PAS'
```

Semantics

A unit-import clause is a list of logical import-contexts, each one optionally associated with its physical

representation denoted by a string whose interpretation is implementation dependent. If the physical representation is absent for an import-context, the compiler must assign the default representation (The default representation may be a file-name derived from the logical name). A unit-include clause is syntactically similar to a unit-import clause except that it lists names of CCNPASCAL source modules. Importing a context makes its items available for use in the importing module, and the compiler ensures that they are used according to the specifications. Including a CCNPASCAL source module has the effect of textually inserting the source module. A module is a closed scope and hence references to items that are either imported or included must be prefixed by the corresponding context or module name (separated by a period ".").

The process of compiling a module M has the following steps:

1. Compile each import-context in the specified order and add its identifiers to the symbol table.
2. Compile each included source-unit in the specified order and add its exported identifiers to the symbol table.

3. Compile the module M.

The imported contexts, the included source-units and the module M form a single scope level. An import-context or an included source-unit need not be self-sufficient regarding the identifiers it refers to. However, during compilation, each identifier that is referenced must be available in the symbol table. Any reference from a unit to an identifier declared in some other unit must be qualified by the name of the unit (separated by a period).

A.4 PROGRAMS

A program is a special kind of module that can be called (invoked) for execution. A program definition may specify a list of formal parameters, actuals for which must be supplied when the program is called.

Syntax

⟨program⟩ ::= program ⟨program-id⟩ ⟨formal-program-param⟩
⟨unit-import⟩ [⟨unit-include⟩] [";"]
[⟨decl-seq⟩] ⟨code-body⟩ ⟨program-id⟩"."

⟨program-id⟩::= ⟨identifier⟩

⟨formal-program-param⟩ ::= [⟨formal-par-list⟩]

⟨code-body⟩ ::= **begin** [⟨stmt-seq⟩] **end**

⟨stmt-seq⟩ ::= {";" ⟨statement⟩} [";"]

⟨call-stmt⟩ ::= **call** ⟨program-id⟩ ⟨actual-program-param⟩

⟨actual-program-param⟩ ::= [⟨actual-par-list⟩]

Examples

```
program PASCAL(INPUT,OUTPUT,LISTING:FILE) import
   FILESYS; {declarations} .
begin
  {code for the Pascal compiler}
end PASCAL.
program USER import PASCAL;
begin
  .
  call PASCAL(SOURCE,BINARY,LIST);
  .
end USER.
```

Semantics

A program is a collection of data and procedure declarations and a main code-body. It must be given a name (denoted by ⟨program-id⟩ that may be used by the

implementation. The main difference between a program and a module is that a program is an entity for invocation while a module is not.

A program definition may specify a list of formal parameters (denoted by ⟨formal-program-param⟩), actuals for which must be supplied when the program is called. A program is called by executing a call statement on its ⟨program-id⟩ and passing a list of actual parameters (denoted by ⟨actual-program-param⟩) which must be equivalent to the corresponding ⟨formal-program-param⟩ A program call must be implemented as a sequential control transfer (and not as a process invocation). A program can be called any number of times and the implementation must ensure that different executions of the program are independent.

A program cannot export any of its variable or routine identifiers. However, it may export constant or type identifiers. Each program is associated with an implicit-context identified by the ⟨program-id⟩ . Compilation of a program also generates a context corresponding to each explicit context declaration appearing in the program text.

However, a program can be referenced from outside only through its implicit-context or by directly including the program text.

A.5 CONTEXT DECLARATIONS

While the implicit-context for a module collects the specifications of all exported identifiers, it is also possible to collect into a context the specifications of a selected set of identifiers with possible restrictions on their access-rights. This is done through the use of a context declaration.

Syntax

⟨context-decl⟩ ::= context ⟨context-id⟩ [";"]
[{";" ⟨context-item⟩ [";"]]
end ⟨context-id⟩

⟨context-id⟩ ::= ⟨identifier⟩

⟨context-item⟩ ::= const ":" {"," ⟨interface-const⟩} !
type ":" {"," ⟨interface-type⟩} !
var ":" {"," ⟨interface-var⟩} !
procedure ":" {"," ⟨interface-proc⟩}
function ":" {"," ⟨interface-func⟩}

⟨interface-const⟩ ::=[⟨identifier⟩"="]⟨const-id⟩ ¦
⟨identifier⟩"=" ⟨const-access⟩

⟨interface-type⟩ ::=[⟨identifier⟩"="]⟨type-id⟩
⟨access-rights⟩ ¦
⟨identifier⟩ "=" ⟨type-access⟩

⟨interface-var⟩ ::= [⟨identifier⟩ "="]⟨var-id⟩
[⟨access-rights⟩] ¦
⟨identifier⟩ "=" ⟨variable⟩

⟨interface-proc⟩::= [⟨identifier⟩ "="]⟨proc-id⟩ ¦
⟨identifier⟩ "=" ⟨proc-access⟩

⟨interface-func⟩::= ⟨identifier⟩ "=" ⟨func-id⟩ ¦
⟨identifier⟩"=" ⟨func-access⟩

Example

The following is a context declaration that exports the same items as the implicit-context IODRIVER of section A.2:

```
context IOCHANNEL;
  type: BUFTYPE;
  procedure: DOIO;
  function: STATUS;
end IOCHANNEL;
```

It may be noted that this is a context declaration, while the one in section A.2 is a generated context whose encoding may be implementation dependent.

A view to the module IODRIVER which allows only the testing of an input/output channel but which does not allow actual input/output operations may be exported by declaring the following (explicit) context in the module IODRIVER:

```
context IOSTATUS;
  function: STATUS;
end IOSTATUS;
```

Semantics

A context declaration can appear only in the outermost block of a compilation unit. When a unit is compiled, corresponding to each context declaration a unit. When a unit is compiled, corresponding to each context declaration a context is generated consisting of the specifications of all identifiers listed in it. However, users of the generated context will have no information about its 'parent' unit. Each item exported in a context declaration can optionally be renamed (the new name is denoted by ⟨identifier⟩ in the syntax, and can possibly be the same

as the old name). When renamed, the specification of the item contains the new name but not the old name.

Identifiers listed in a context declaration must be categorized as constants, types, variables, procedures or functions. They must either be formally declared (their declarations need not textually precede the context declaration) in the outermost block of the compilation unit or be imported into it from other contexts. In either case, the categorization of an identifier in a context declaration must correspond to its actual definition.

An item denoting a type or a variable can optionally specify an access-rights field in which case the specification of the identifier in the generated context provides for only those access-rights,. If an item is an indexed array, each array index must be a ⟨constant⟩ . The generated context is named after the ⟨context-id⟩ used in the context declaration.

A ⟨context-id⟩ cannot be used in more than one context declaration. Optionally, a ⟨module-id⟩ or ⟨program-id⟩ itself may be used as a ⟨context-id⟩ in a context declaration. When used, the resulting context is merely appended to the

implicit-context. This feature is useful in re-exporting through the implicit-context some of the identifiers that are imported (or included) into the compilation unit.

APPENDIX 2

INSTRUCTIONS FOR USE

The simulator is mainly meant for interactive users, who can run the simulator by the command:

.R SIM16 CR

The simulator identifies itself and comes with the prompt character '#', ready to accept user commands at the character level. A detailed description of the simulator commands is given in the following pages. Responses from the simulator (other than the prompt character) are underlined.

nA

If n (an octal number) is non-null, inserts a break for the address n. Atmost 8 address-breaks are allowed. Address-breaks for references in an instruction are signalled only after the execution of the instruction.

If n is null, removes all address-breaks.

$ nA

The status of the n th (only the rightmost digit

considered) address-break is displayed. The n th address-break is free if the status is 777777; else the status gives the address where the break was inserted. By default, n=0.

#0$ nA

Removes the n th address-break, if it exists. By default, n=0.

nB

If n is non-null, inserts a break for the instruction at PC value n. Atmost 8 instruction-breaks are allowed. Instruction-breaks are signalled before the execution of the instruction.

If n is null, removes all instruction-breaks.

$ nB

The status of the n th instruction-break is displayed. The n th instruction-break is free if the status is 777777; else the status gives the address where the break was inserted. By default, n=0.

#0$ nB

Removes the n th instruction-break, if it exists. By default, n=0.

C

Does the function of the continue-switch on the console. Brings the terminal into NOECHO-mode and continues execution from the current PC value.

#mDn CR

If n is non-null, dumps memory from octal address m to n (both inclusive) onto the current HSP-file in absolute binary format as one block. Any number of blocks can be dumped onto the same HSP-file one after the other.

If n is null, dumps a transfer-block with m as the starting address and the current HSP-file is closed.

nE

Closes any existing HSR/HSP-files. If n is non-null, types out instruction - counts for all TDC-316 instructions. In any case, types out the simulated time and the simulator-time in milli-seconds, and also the total

number of instructions executed and the average execution time per instruction. Finally, exits from the simulator.

nG

Does the function of the start-switch on the console (but does not automatically initialize the system). Brings the terminal into NOECHO-mode and starts execution from octal address n (instead of loading PC from switch-register). By default, n=0.

I

Does the function of the console <u>init</u>. Brings the system to the power-on state. Initializes all trap-conditions, real-time, device registers, CPU status and stack-limit registers, event and interrupt queues and ODT status.

L File: dev : file. ext [ppn] <prot>

Loads the specified absolute binary file into the simulated-core. Starting address, as specified by the transfer-block, is loaded into PC. Any number of files can be loaded with successive L-commands.

nM

If n is non-null, takes n as the new mask for word search.

If n is null, displays the current mask value.

N

If the system was in the single instruction mode, brings it into the normal mode. Otherwise it has no effect.

P File: dev : file . ext [ppn] <prot>

Creates a binary HSP-file with the given name. If a file with that name already exists and is not write-protected, that file is over-written. Only one HSP-file can be created at a time. When a new file is created any old output file is automatically closed. HSP-files are not saved unless they are closed. E-command closes any existing HSP-file.

OP File: dev : file . ext [ppn] <prot>

Creates an ASCII HSP-file with the given name. Rules are same as for binary files.

R File: dev : file . ext [ppn] <prot>

Opens the specified binary HSR-file. Only one HSR-file can be kept open at any time. When a new file is opened, any old file that is open is automatically closed. If the specified file is not accessible, error is signalled to the user. An HSR-file need not be closed for it to be saved. However, E-command closes any existing HSR-file.

OR File: dev : file ↑ ext [ppn] <prot>

Opens the specified ASCII HSR-file. Rules are same as for binary files.

S

This simulates the single-instruction mode switch on the console. If the system was in the normal mode, brings it into the single-instruction mode. Otherwise, it has no effect. In the single-instruction-mode, execution of an instruction causes a status dump onto the user's terminal. The status consists of:

Nemnic : BRASS mnemonic for the instruction
PCval : address of the instruction

Instr : the instruction itself

SOurce: source operand(zero if none)

DstAdd: destination address(zero if none)

MQval : MQ-register value

SPval : stack-pointer value

PSval : CPU status

SCval : shift-counter value

Instruction-break, address-break, halt and wait are not operative in the single-instruction-mode.

T

Types-out the simulated-time elasped since the last T-command in the format, hrs : mts : sec : msecs.

nW

Searches the memory for the word n after masking with word-search-mast. If the search is successful, address and contents of the word are typed-out in the format address "/" contents. In case the search fails, it is signalled to the user. By default, n=0.

m / or \m

Opens the word or byte addressed by m. It will

be clear from the context whether the currently opened location is a word or byte. When location m is opened its contents are typed and the system awaits user response. The user may open another location or he may type-in an octal value n followed by a special character.

If n is non-nul, the location m is modified with n. The special character can be one of the following:

/	reopen the last word opened
\	reopen the last byte opened
LF	open the following location
CR	close the opened location
(^)	open the previous location
_ (under score)	open the relatively addressed word (illegal for bytes)
@	open the absolutely addressed word (illegal for bytes)
	open the word to which the branch refers (illegal for bytes)
	open the next location of previous sequence

Any time the simulated CPU is active, it can be halted by typing-in control X from the KED. Whenever the control is given to the user, the terminal is restored to ECHO-mode.

A list of TDC-316 register address is given below:

R0:	177740	R4:	177750
R1:	177742	R5:	177752
R2:	177744	R6:	177754
R3:	177746	R7:	177758

Program status word :	177700
Switch - Register :	177702
Stack-Limit Register :	177704
Shift Counter :	177734
MQ Register :	177736

HSR Status :	177660	HSR Buffer :	177662
HSP Status :	177664	HSP Buffer :	177666
KBD Status :	177670	KBD Buffer :	177672
TTY Status :	177674	TTY Buffer :	177676
CLK Status :	177310		

```
  INCLUDE ICSYS;
  CONST DEBUGGING=FALSE;
  VAR TRAPFLAG:BOOLEAN;
  VAR STRAPFLAG:BOOLEAN;
  VAR BTRAPFLAG:BOOLEAN;
  VAR RTRAPFLAG:BOOLEAN;
  VAR BUSTRAPFLA:BOOLEAN;
  VAR ILLTRAPFLA:BOOLEAN;
  VAR HALTFLAG:BOOLEAN;
  VAR BYTEFLAG:BOOLEAN;
  VAR WAIT:BOOLEAN;
  VAR REGMODE:BOOLEAN;
  VAR RUNMODE:BOOLEAN;
  VAR SPMODE:BOOLEAN;
  VAR HSRMODE:BOOLEAN;
  VAR HSPMODE:BOOLEAN;
  VAR REALTIME:INTEGER;
  TYPE TRAPTYPE=(ITRAP,STRAP,BTRAP,RTRAP,ILLTRAP,BUSTRAP);
  TYPE MODETYPE=(ZREGMODE,ZRUNMODE,ZSPMODE,ZHSRMODE,ZRSPMODE);
  TYPE FLAGTYPE=(ZHALTFLAG,ZBYTEFLAG,ZWAIT);
  PROCEDURE SETTRAP(NEWVALUE:BOOLEAN;LOCATION:TRAPTYPE);
  PROCEDURE SETMODE(NEWVALUE:BOOLEAN;MODE:MODETYPE);
  PROCEDURE SETFLAG(NEWVALUE:BOOLEAN;FLAG:FLAGTYPE);
  PROCEDURE SETTIME(NEWVALUE:INTEGER);
END GLOBAL.
```

```
00200     INCLUDE MEMORY;
00300     PROCEDURE ICLOSE;
00400     PROCEDURE OCLOSE;
00500     PROCEDURE IFILE(FILENAME:STRING(9);OCTFLAG:BOOLEAN);
00600     PROCEDURE OFILE(FILENAME:STRING(9);OCTFLAG:BOOLEAN);
00700     PROCEDURE LOADFILE(OCTFLAG:BOOLEAN;FILENAME:STRING(9));
00800     PROCEDURE DUMP(DBEGIN:INTEGER;DEND:INTEGER;OCTFLAG:BOOLEAN);
00900   END FILEOP.
```

```
INCLUDE MEMORY;
VAR CLKSTATUS:INTEGER;
VAR KBDSTATUS:INTEGER;
VAR HSRSTATUS:INTEGER;
VAR HSPSTATUS:INTEGER;
VAR TTYSTATUS:INTEGER;
VAR KBDBUF:INTEGER;
VAR HSRBUF:INTEGER;
VAR HSPBUF:INTEGER;
VAR TTYBUF:INTEGER;
VAR KBDCHR:CHAR;
CONST HSRPRTY=1;
CONST HSPPRTY=1;
CONST KBDPRTY=1;
CONST TTYPRTY=1;
CONST CLKPRTY=6;
CONST HSRLOCATIO=48;
CONST HSPLOCATIO=52;
CONST KBDLOCATIO=56;
CONST TTYLOCATIO=60;
CONST CLKLOCATIO=200;
CONST STKLOCATIO=4;
CONST TRPLOCATIO=8;
CONST BPTLOCATIO=12;
CONST EMTLOCATIO=16;
CONST IOTLOCATIO=20;
CONST RESLOCATIO=24;
CONST ILLBUS=28;
CONST IOBUFSIZE=600;
CONST HSR=65456;
CONST HSP=65460;
CONST KBD=65464;
CONST TTP=65468;
CONST CLK=65224;
CONST HSR1=65457;
CONST HSR2=65458;
CONST HSR3=65459;
CONST HSP1=65461;
CONST HSP2=65462;
CONST HSP3=65463;
CONST KBD1=65465;
CONST KBD2=65466;
CONST KBD3=65467;
CONST TTP1=65469;
CONST TTP2=65470;
CONST TTP3=65471;
CONST CLK1=65225;
CONST EQSIZE=25;
CONST IQSIZE=25;
PROCEDURE INITEVENTI;
PROCEDURE INITSTATUS;
PROCEDURE BITSTATUSS(BITNUM:INTEGER;NEWVALUE:BOOLEAN;LOCATION:INTEGER);
PROCEDURE MCLRFALS;
PROCEDURE ENTEREVENT(DEV:INTEGER;TIME:INTEGER;PRTY:INTEGER);
PROCEDURE ENTERINTER(INTVEC:INTEGER;PRTY:INTEGER);
PROCEDURE IODONE(EVENT:INTEGER);
PROCEDURE SEREVENT;
PROCEDURE SERTRAP(LOCATION:INTEGER);
PROCEDURE SERINTERRU;
```

```
  PROCEDURE IOREAD(ADDRESS:INTEGER;VAR TEMP:INTEGER);
  PROCEDURE IORITE(ADDRESS:INTEGER;DATA:INTEGER);
  PROCEDURE KBDSIM;
  PROCEDURE CLKSIM;
END REALS.
```

```
  INCLUDE REALS,DEKODE;
  CONST MAXINS=76;
  CONST MAXBREAK=7;
  VAR ADRFLAG:BOOLEAN;
  VAR INSFLAG:BOOLEAN;
  VAR SINGLE:BOOLEAN;
  VAR ADRBREAK:ARRAY [0..7] OF INTEGER;
  VAR INSBREAK:ARRAY [0..7] OF INTEGER;
  VAR INSCOUNT:ARRAY [0..76] OF INTEGER;
  VAR NEMNIC:ARRAY [0..76] OF STRING(5);
  TYPE BREAKTYPE=(ZADRBREAK,ZINSBREAK);
  TYPE BRKFLAG=(ZINSFLAG,ZADRFLAG,ZSINGLE);
  PROCEDURE INITINSCOU;
  PROCEDURE SETBREAK(BREAK:BREAKTYPE;BREAKNUM:INTEGER;NEWVALUE:INTEGER);
  PROCEDURE SETBRKFLAG(FLAG:BRKFLAG;NEWVALUE:BOOLEAN);
  PROCEDURE SIMULATE;
END SIMRUN.
```

```
CONTEXT MODULE DEKODE;
  INCLUDE EXEC;
  TYPE AREATYPE=(SOURCE,DESTINATIO);
  VAR DSTTIME:INTEGER;
  VAR SRCTIME:INTEGER;
  PROCEDURE INITTIME;
  FUNCTION DECODE(INS:INTEGER) RETURN INTEGER;
  PROCEDURE OPERANDS(INSREG:INTEGER;AREA:AREATYPE;VAR ADD:INTEGER);
END DEKODE.
```

```
  INCLUDE UTILTY,GLOBAL,UNI;
  PROCEDURE INITIDCMEM;
  PROCEDURE MREADWORD(ADDRESS:INTEGER;VAR REQWORD:INTEGER);
  PROCEDURE MREADBYTE(ADDRESS:INTEGER;VAR REQBYTE:INTEGER);
  PROCEDURE MRITEWORD(ADDRESS:INTEGER;DATA:INTEGER);
  PROCEDURE MRITEBYTE(ADDRESS:INTEGER;DATA:INTEGER);
  PROCEDURE READWORD(ADDRESS:INTEGER;VAR REQWORD:INTEGER);
  PROCEDURE READBYTE(ADDRESS:INTEGER;VAR REQBYTE:INTEGER);
  PROCEDURE WRITEWORD(ADDRESS:INTEGER;DATA:INTEGER);
  PROCEDURE WRITEBYTE(ADDRESS:INTEGER;DATA:INTEGER);
END MEMORY.
```

```
INCLUDE UTILITY,REALS;
VAR MQVAL:INTEGER;
VAR SLVAL:INTEGER;
VAR SRVAL:INTEGER;
VAR PSVAL:INTEGER;
VAR SCVAL:INTEGER;
VAR REG:ARRAY [0..15] OF INTEGER;
TYPE PROCEZR=(ZREADWORD,ZREADBYTE,ZWRITEWORD,ZWRITEBYTE);
TYPE SUBPROCEZR=(ZCPU,ZCORE,ZINOUT,ZBUSERR,ZODDADR);
CONST PC=65504;
CONST SP=65506;
CONST MEMSIZE=14336;
CONST PS=65472;
CONST SWR=65474;
CONST SLR=65476;
CONST SC=65500;
CONST MQR=65502;
CONST SWR1=65475;
CONST MQR1=65503;
CONST SLR1=65477;
PROCEDURE INITREGS;
PROCEDURE ERRORS(FROMPROC:PROCEZR;SUBPROC:SUBPROCEZR;ADDRESS:INTEGER);
PROCEDURE SETMASKPR(LOCATION:INTEGER;NEWVALUE:INTEGER);
PROCEDURE WDCPUREAD(ADDRESS:INTEGER;VAR REQWORD:INTEGER);
PROCEDURE BTCPUREAD(ADDRESS:INTEGER;VAR REQBYTE:INTEGER);
PROCEDURE WDCPUWRITE(ADDRESS:INTEGER;DATA:INTEGER);
PROCEDURE BTCPUWRITE(ADDRESS:INTEGER;DATA:INTEGER);
PROCEDURE WDIOREAD(ADDRESS:INTEGER;VAR REQWORD:INTEGER);
PROCEDURE BTIOREAD(ADDRESS:INTEGER;VAR REQBYTE:INTEGER);
PROCEDURE WDIOWRITE(ADDRESS:INTEGER;DATA:INTEGER);
PROCEDURE BTIOWRITE(ADDRESS:INTEGER;DATA:INTEGER);
PROCEDURE REGSET(NEWVALUE:INTEGER;REGNUM:INTEGER);
PROCEDURE ADRUNIBUS(ADDRESS:INTEGER;VAR BLOCK:SUBPROCEZR)
END UNI.
```

```
CONTEXT MODULE EXEC;
  INCLUDE MEMORY;
  VAR SRC:INTEGER;
  VAR DSTADD:INTEGER;
  VAR INSTIME:INTEGER;
  VAR COZSB:INTEGER;
  VAR SUBREG:INTEGER;
  VAR OFFSET:INTEGER;
  TYPE ADRTYPE=(ZSRC,ZDSTADD,ZCOZSB,ZSUBREG,ZOFFSET);
  TYPE TYPEOPCODE=(JUNK,IOR,IORB,CMP,CMPB,AND,ANDB,MUL,DIVX,MOV,
                   MOVB,ADD,SUB,CLB,CLBB,CAL,CLR,CLRB,TST,TSTB,
                   INC,INCB,CAD,CADB,DEC,DECB,CSB,CSBB,COM,COMB,
                   NOTS,NOTB,SHR,SHRB,ESR,ELR,SHL,SHLB,ELL,ENM,
                   CIR,CIRB,CIL,CILB,ECL,ECR,JMP,EXB,DTN,TNE,
                   TGE,TGT,TRA,TEQ,TLT,TLE,TPL,TOC,TCC,THI,
                   TMI,TOS,TCS,TLS,SVC,XPR,TRP,SETCC,CLRCC,RES,
                   HLT,WFI,REI,BPT,MCL);
  PROCEDURE INITOPERAN;
  PROCEDURE SETADR(NEWVALUE:INTEGER;LOCATION:ADRTYPE);
  PROCEDURE EXECUTE(OPCODE:TYPEOPCODE);
END EXEC.
```

START User KRISHNA [70,103] Job DIALOG Seq. 1647 Date 10-May-82 18:05:13

START User KRISHNA [70,103] Job DIALOG Seq. 1647 Date 10-May-82 18:05:13

```
K   K  RRRR    III    SSSS  H   H  N   N   AAA
K  K   R   R    I    S      H   H  NN  N  A   A
K K    R   R    I    S      H   H  N N N  A   A
KK     RRRR     I     SSS   HHHHH  N N N  A   A
K K    R R      I        S  H   H  N  NN  AAAAA
K  K   R  R     I        S  H   H  N   N  A   A
K   K  R   R   III   SSSS   H   H  N   N  A   A
```

START User KRISHNA [70,103] Job DIALOG Seq. 1647 Date 10-May-82 18:05:13

START User KRISHNA [70,103] Job DIALOG Seq. 1647 Date 10-May-82 18:05:13

START User KRISHNA [70,103] Job DIALOG Seq. 1647 Date 10-May-82 18:05:13

START User KRISHNA [70,103] Job DIALOG Seq. 1647 Date 10-May-82 18:05:13

START User KRISHNA [70,103] Job DIALOG Seq. 1647 Date 10-May-82 18:05:13

START User KRISHNA [70,103] Job DIALOG Seq. 1647 Date 10-May-82 18:05:13

START User KRISHNA [70,103] Job DIALOG Seq. 1647 Date 10-May-82 18:05:13

```
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000011111111112222222222333333333344444444445555555555666666666677777777
1234567890123456789012345678901234567890123456789012345678901234567890123456
```

```
00100
00200   program DIALOG options(VERSION = NSIM, DEFAULTEXT)
00300     include FILEOP, SIMRUN, UNI, UTILTY, GLOBAL, IOSYS;
00400     type
00500       CHARSET = set of char;
00600     var
00700       DEVICE: string(6);
00800       FILENAME: string(9);
00900       LEFTPPN, RIGHTPPN, PROJECTION, MODE, DBEGIN, DEND:
01000       integer;
01100       OCT, OLDADDRESS, TIMER, LAPS, I, SEARCHMASK, REQWORD,
01200       TIME, TIME1:
01300       integer;
01400       DECTIME: integer;
01500       CONDITION, OCTFLAG, ARCNGEILE: boolean;
01600       CH, CHA, CR, LF: char;
01700       D: string(9);
01800
01900     procedure INITIALIZE;
02000       %*********************************\
02100       begin
02200         DECTIME := 0;
02300         %INITIALIZES THE TIME TAKEN FOR SIMULATION \
02400         SETTIME(0); INITTIME;  %declared in DEKODE\
02500         for I in TRAP to BUSTRAP
02600           do SETTRAP(false, I);
02700         for I in ZWAIT to ZHALTFLAG
02800           do SETFLAG(false, I);
02900         for I in ZREGMODE to ZHSPMODE
03000           do SETMODE(false, I);
03100         %declared in GLOBAL\
03200         for I in ZINSFLAG to ZSINGLE
03300           do SETBRKFLAG(I, false);
03400         INITEVENTINTS;
03500         INITSTATUSBUF;  %declared in REALS \
03600         INITINSCOUNT;  %declared in SIMRUN\
03700         INITREGS;  %declared in UNI  \
03800         INITTDCMEM;
03900         INITOPERANDS;
04000         ICLOSE; OCLOSE;
04100         for I in 0 to MAXBREAK do
04200         begin
04300           SETBREAK(ZADRBREAK, I, 0);
04400           SETBREAK(ZINSBREAK, I, 0)
04500         end
04600       end INITIALIZE;
04700
04800
04900     procedure REPORT(FLAG:boolean);
05000       %*****************************************************
05100       ****\
05200       % reports the simulated time,instruction counts,total nu
05300       mber of in
05400       structions and average execution time \
05500       var
05600         TOTAL, RTIME, TEMP, I: integer;
05700       begin
05800         TOTAL := 0; TEMP := 0;
05900         RTIME := REALTIME div 10;
06000         ICLOSE; OCLOSE;
```

```
06100        WRITE('TDC16 time in micro seconds:');
06200        WRITELN(RTIME);
06300        WRITE('DEC10 time in micro seconds: ');
06400        WRITELN(DECTIME * 1000);
06500        %because RUNTIME system routine gives time in milli se_
06600        conds\
06700        WRITELN('Simulation Speed Ratio is 1/', (DECTIME *
06800
06900
07000
07100                1000) div
07200                RTIME);
07300        if FLAG
07400          then
07500        begin
07600          WRITELN(' Instruction Counts follow '); WRITELN;
07700          for I in 0 to MAXINS
07800            do WRITE('      ', MEMNIC[I + 1], INSCOUNT[I])
07900        end;
08000        for I in 0 to MAXINS
08100          do TOTAL := TOTAL + INSCOUNT[I];
08200        WRITELN;
08300        WRITELN('Total number of instructions is ', TOTAL);
08400        WRITELN('Average simulated time per instruction is ',
08500                RTIME, '/'
08600                , TOTAL, '     in  micro seconds');
08700        WRITELN(
08800        'Average simulated time per instruction on DEC10 is ',
08900                DECTIME * 1000, '/', TOTAL);
09000      end REPORT;
09100    procedure FILESPEC;
09200      %*****************************************************\
09300      % reads input output file specifications from kbd in th
09400      e format:
09500      dev:file.ext[lppn,rppn]<prot> \
09600      var
09700        I, NUMVALUE: integer;
09800        NAME: string(6);
09900        CH: char;
10000      procedure GETNAME(var CH:char);
10100        %=================================================\
10200        %  gets a <=6 character name followed by special chara
10300        cter \
10400        var
10500          I: integer;
10600        begin
10700          for I in 1 to 6
10800            do NAME[I] := ' ';
10900          CH := ' '; I := 0;
11000          while CH = ' '
11100            do READ(CH);
11200          while (CH in CHARSET['0' .. '9']) or (CH in CHARSET[
11300                'A' .. 'Z'
11400                ]) or (CH in CHARSET['a' .. 'z'])
11500                  do
11600          begin
11700            I := I + 1;
11800            if I <= 6
11900              then NAME[I] := CH;
12000            READ(CH)
```

```
12100
12200
12300
12400
12500
12600          end;
12700          if CH = CHR(033B)
12800            then WRONGFILE := true;
12900          ;
13000          while CH = ' '
13100            do READ(CH)
13200        end GETNAME;
13300      procedure GETNUMBER(var NUMBER:integer; var CH:char);
13400        %==========================================================
13500        ======%===
13600        ===\
13700        % gets an octal number follwed optionally by a special
13800        charactor
13900        \
14000        const
14100          RADIX = 8;
14200        begin
14300          READ(CH);
14400          while (CH = ' ')
14500            do READ(CH);
14600          NUMBER := 0;
14700          while (CH in CHARSET['0' .. '7'])
14800                do
14900          begin
15000            NUMBER := NUMBER * RADIX + ORD(CH) - ORD('0');
15100            READ(CH)
15200          end;
15300          if CH in CHARSET('8' .. '9') then
15400          begin
15500            WRITELN( % BELL,\' PPN IS NOT AN OCTAL');
15600            WRONGFILE := true
15700          end;
15800          if CH = CHR(033B)
15900            then WRONGFILE := true
16000        end GETNUMBER;
16100      begin % FILESPEC \
16200        FILENAME := '          ';
16300        WRITE('      FILE:');
16400        WRONGFILE := false;
16500        GETNAME(CH);
16600        if CH = CHR(015B) then
16700          for I in 1 to 6
16800            do FILENAME[I] := NAME[I];
16900        while CH in CHARSET[':', '.', '[', ']', '<', '>']
17000              do
17100        case CH of
17200          ':':
17300            begin
17400              for I in 1 to 6
17500                do DEVICE[I] := NAME[I];
17600
17700                        2
17800
17900
18000
```

```
                GETNAME(CH);
                if (CH = '.') or (CH = '[') then
                  for I in 1 to 6
                    do FILENAME[I] := NAME[I]
              end;
            '.':
              begin
                for I in 1 to 6
                  do FILENAME[I] := NAME[I];
                GETNAME(CH);
                for I in 1 to 3
                  do FILENAME[I + 6] := NAME[I]
              end;
            '[':
              begin
                GETNUMBER(NUMVALUE, CH);
                LEFTPPN := NUMVALUE;
                if CH = ','
                  then
                begin
                  GETNUMBER(NUMVALUE, CH);
                  RIGHTPPN := NUMVALUE
                end
                else
                begin
                  WRITELN( % BELL,\
' COMMA MISSING IN PPN OR PPN IS NOT AN OCTAL NUMBER')
                  WRONGFILE := true;
                  return
                end
              end;
            ']':
              if not (EOLN)
                then READ(CH)
              else return;
            '<':
              begin
                GETNUMBER(NUMVALUE, CH);
                PROTECTION := NUMVALUE
              end;
            '>': return
          end;
        if CH = CHR(033B)
          then WRONGFILE := true;
        ;
        if FILENAME = '      '
          then WRONGFILE := true
      end FILESPEC;
```



```
  procedure GETOCT(var SEPERATOR:char; var OCTFLAG:boolean;
                   var OCT:integer);
    %*************************************************************
    **********
     ****************\
```

```
24100      const
24200        RADIX = 8;
24300      var
24400        CH: char;
24500        DOAGAIN: boolean;
24600      begin
24700        OCT := 0;
24800        OCTFLAG := false; DOAGAIN := true;
24900        while DOAGAIN do
25000        begin
25100          READ(CH);
25200          while CH = ' '
25300            do READ(CH);
25400          while (CH in CHARSET['0' .. '7']) and not EOF
25500                do
25600          begin
25700            OCTFLAG := true;
25800            OCT := OCT * RADIX + ORD(CH) - ORD('0');
25900            READ(CH)
26000          end;
26100          SEPERATOR := CH;
26200          if CH in CHARSET['8' .. '9']
26300            then
26400          begin
26500            WRITELN(^G,\
26600            ' ? NOT AN OCTAL NUMBER TYPE IT AGAIN ');
26700            OCT := 0
26800          end
26900          else DOAGAIN := false
27000        end % while \
27100      end GETOCT;
27200
27300    procedure DEBUG(CHA:char; ADDRESS:integer);
27400      %*************************************************
27500      ********\
27600      type
27700        VIEWTYPE = (DECWORD, IOCWORD);
27800        WORDTYPE =
27900          packed record
28000            case VIEWTYPE of
28100              DECWORD: (FULLWORD: integer);
28200              IOCWORD: (DUMMY: 0 .. 17B;
28300                        LEFTWORD, RIGHTWORD: 0 .. 177777B)
28400          end WORDTYPE;
28500      var
28600
28700                                  - 4 -
28800
28900
29000
29100        OLDCHA, TEMPCHA: char;
29200        VALUE, REGWORD, REGBYTE, STEP, TEMP, TEMP1, EXBYTE,
29300        EXWORD,
29400        TEMPADDRESS: integer;
29500        SEQFLAG: boolean;
29600        R1: WORDTYPE(DECWORD);
29700        R2: WORDTYPE(IOCWORD);
29800      begin
29900        TEMPCHA := CHA; TEMPADDRESS := ADDRESS;
30000        REGWORD := 0; REGBYTE := 0; STEP := 0; TEMP := 0;
```

```
30100          SEQFLAG := true;
30200          loop
30300            case TEMPCHA of
30400              '/':
30500                begin
30600                  if (ADDRESS mod 2) <> 0
30700                    then
30800                  begin
30900                    WRITE( & BELL,\'?? ODD ADDRESS  ');
31000                    return
31100                  end;
31200                  if SEQFLAG
31300                    then
31400                  begin
31500                    OLDCHA := TEMPCHA;
31600                    OLDADDRESS := TEMPADDRESS
31700                  end;
31800                  READWORD(TEMPADDRESS, VALUE);
31900                  WRITELN(C: 6, VALUE);
32000                  GETOCT(TEMPCHA, OCTFLAG, OCT);
32100                  if OCTFLAG
32200                    then
32300                  begin
32400                    VALUE := OCT;
32500                    if (TEMPCHA = '/') or (TEMPCHA = '\')
32600                      then TEMPADDRESS := VALUE
32700                    else WRITEWORD(TEMPADDRESS, VALUE)
32800                  end;
32900                  STEP := 2
33000                end;
33100              '\':
33200                begin
33300                  if SEQFLAG
33400                    then
33500                  begin
33600                    OLDCHA := TEMPCHA;
33700                    OLDADDRESS := TEMPADDRESS
33800                  end;
33900                  READBYTE(TEMPADDRESS, VALUE);
34000                  WRITELN(C: 6, VALUE);
34100
34200                                5
34300
34400
34500
34600                  GETOCT(TEMPCHA, OCTFLAG, OCT);
34700                  if OCTFLAG
34800                    then
34900                  begin
35000                    if (TEMPCHA = '\') or (TEMPCHA = '/')
35100                      then TEMPADDRESS := OCT
35200                    else WRITEBYTE(TEMPADDRESS, OCT)
35300                  end;
35400                  STEP := 1
35500                end;
35600              '^':
35700                begin
35800                  GETWORD(TEMPADDRESS - STEP, RIGHT, TEMPADDRESS
35900                          );
36000                  TEMPCHA := OLDCHA;
```

```
361000                WRITE(O: 6, TEMPADDRESS, TEMPCHA)
362000              end;
363000            '':
364000              begin
365000                if OLDCHA = '\'
366000                  then
367000                begin
368000                  WRITE( % BELL,\
369000
370000    '?   CAN NOT OPEN RELATIVELY ADDRESSED -BYTE-');
371000                  return
372000                end;
373000                if SEQFLAG
374000                  then
375000                begin
376000                  OLDADDRESS := TEMPADDRESS;
377000                  SEQFLAG := false
378000                end;
379000                GETWORD(TEMPADDRESS + VALUE + 2, RIGHT,
380000                        TEMPADDRESS);
381000                WRITE(O: 6, TEMPADDRESS, '  ');
382000                TEMPCHA := '/'
383000              end;
384000            '@':
385000              begin
386000                if OLDCHA = '\'
387000                  then
388000                begin
389000                  WRITELN( % BELL,\
390000                   '?  CAN NOT OPEN ABSOLUTELY ADDRESSED-BYTE-'
391000                        );
392000                  return
393000                end;
394000                if SEQFLAG
395000                  then
396000
397000                                6
398000
399000
400000
401000              begin
402000                OLDADDRESS := TEMPADDRESS;
403000                SEQFLAG := false
404000              end;
405000              TEMPADDRESS := VALUE;
406000              WRITE(O: 6, TEMPADDRESS, '  ');
407000              TEMPCHA := '/'
408000            end;
409000          '>':
410000            begin
411000              if OLDCHA = '\'
412000                then
413000              begin
414000                WRITELN( % BELL,\'?  ');
415000                return
416000              end;
417000              if SEQFLAG
418000                then
419000              begin
420000                OLDADDRESS := ADDRESS;
```

```
                SEQFLAG := false
                end;
              EXTEND(TEMPADDRESS, WORDS, EXWORD);
              TEMP1 := EXWORD;
              EXTEND(VALUE, BYTE, EXBYTE);
              GETWORD(TEMP1 + 2 * EXBYTE + 2, RIGHT,
                      TEMPADDRESS);
              WRITE(O: 6, TEMPADDRESS, '    ');
              TEMPCHA := '/'
            end;
          '<':
            begin
              SEQFLAG := true;
              GETWORD(OLDADDRESS + STEP, RIGHT, TEMPADDRESS)
              ;
              TEMPCHA := OLDCHA;
              WRITE(O: 6, TEMPADDRESS, TEMPCHA, '  ')
            end;
          ' ': exit;
          otherwise:
            if (TEMPCHA = LF)
              then
            begin
              GETWORD(TEMPADDRESS + STEP, RIGHT, TEMPADDRESS
                      );
              TEMPCHA := OLDCHA;
              WRITE(O: 6, TEMPADDRESS, TEMPCHA, '  ')
            end
            else
              if (TEMPCHA <> CR)
```



```
                then
              begin
                WRITELN( % BELL,\'?? ILLEGAL CHARACTOR  ');
                return
              end
              else return
        end % CASE \
      end % LOOP \
    end DEBUG;
  begin %DIALOG\
    CR := CHR(015B); LF := CHR(012B);
    CHARMODE;
    CH := ' '; OCT := 0; OCTFLAG := false; TIMER := 0;
    WRONGFILE := false; DECTIME := 0;
    WRITELN;
    WRITELN('    SIMULATED IDC1.6 :CONPASCAL VERSION');
    INITIALIZE:
    loop
      WRITELN; WRITE(' ');
      GETOCT(CH, OCTFLAG, OCT);
      case CH of
        'A', 'a':
          begin
            if OCTFLAG
              then
```

```
48100            begin
48200              CONDITION := true;
48300              for I in 0 to MAXBREAK
48400                  do
48500                if (ADRBREAK[I] = 0) and (CONDITION)
48600                  then
48700                begin
48800                  SETBRKFLAG(ZADRFLAG, true);
48900                  SETBREAK(ZADRBREAK, I, OCT);
49000                  CONDITION := false
49100                end;
49200              if CONDITION
49300                then WRITELN( % BELL,\
49400                     '? NO. OF ADDRESS BREAKS EXCEEDED 8')
49500            end
49600            else
49700            begin
49800              for I in 0 to MAXBREAK
49900                do SETBREAK(ZADRBREAK, I, 0);
50000              SETBRKFLAG(ZADRFLAG, false)
50100            end
50200          end;
50300        'B', 'b':
50400          begin
50500            if OCTFLAG
50600
50700                                  8
50800
50900
51000
51100              then
51200            begin
51300              CONDITION := true;
51400              if (OCT mod 2) <> 0
51500                then
51600              begin
51700                WRITELN( % BELL,\
51800                ' ILLEGAL INSTRUCTION ADDRESS');
51900                CONDITION := false
52000              end;
52100              if CONDITION
52200                then
52300              begin
52400                for I in 0 to MAXBREAK
52500                    do
52600                  if (INSBREAK[I] = 0) and CONDITION
52700                    then
52800                  begin
52900                    SETBRKFLAG(ZINSFLAG, true);
53000
53100                    SETBREAK(ZINSBREAK, I, OCT);
53200                    CONDITION := false
53300                  end;
53400                if CONDITION
53500                  then WRITELN( % BELL,\
53600
53700  '? NO. OF INSTRUCTION BREAKS EXCEEDED8');
53800              end
53900            end
54000            else
```

```
54100                 begin
54200                   for I in 0 to MAXBREAK
54300                     do SETBREAK(ZINSBREAK, I, 0);
54400                   SETBRKFLAG(ZINSFLAG, false)
54500                 end
54600               end;
54700             'C', 'c':
54800               begin
54900                 if not SINGLE
55000                   then WRITELN;
55100                 SETMODE(true, ZRUNMODE);
55200                 NOECHO;
55300                 TIME1 := RUNTIME;
55400                 TESTMODE;
55500                 % RUNTIME RETURNS THE TIME ELAPSED BETWEEN TWO
55600                 SUCCESSIVE
55700                  CALLS OF THIS PROCEDURE  \
55800                 SIMULATE;
55900                 CHARMODE;
56000                 TIME := RUNTIME - TIME1;
56100
                                    9
56300
56400
56500
56600                 DECTIME := DECTIME + TIME;
56700                 ECHO
56800               end;
56900             'D', 'd':
57000               begin
57100                 DBEGIN := CCI;
57200                 WRITELN('Specify End address');
57300                 GETOCT(CH, OCTFLAG, DEND);
57400                 DUMP(DBEGIN, DEND, OCTFLAG)
57500               end;
57600             'E', 'e': REPORT(OCTFLAG);
57700             % FINISHES THE JOB  \
57800             'G', 'g':
57900               begin
58000                 if OCTFLAG
58100                   then REGSET(CCI, 0);
58200                 WRITELN;
58300                 SETMODE(true, ZRUNMODE);
58400                 NOECHO;
58500                 TIME1 := RUNTIME;
58600                 TESTMODE;
58700                 SIMULATE;
58800                 CHARMODE;
58900                 TIME := RUNTIME - TIME1;
59000                 DECTIME := DECTIME + TIME;
59100                 ECHO
59200               end;
59300             'I', 'i': INITIALIZE;
59400             'L', 'l':
59500               begin
59600                 CH := ' ';
59700                 FILESPEC;
59800                 if not WRONGFILE
59900                   then LOADFILE(OCTFLAG, FILENAME)
60000                   else WRITELN('??', BELL)
```

```
60100            end;
60200          'M', 'm':
60300            if OCTFLAG
60400              then GETWORD(OCT, RIGHT, SEARCHMASK)
60500            else WRITELN(0: 6, SEARCHMASK);
60600          'N', 'n': SETBRKFLAG(ZSINGLE, false);
60700          'P', 'p':
60800            begin
60900              SETMODE(OCTFLAG, ZHSRMODE);
61000              DEVICE := '      '; FILENAME := '          ';
61100              LEFTPPN := 0; RIGHTPPN := 0; PROTECTION := 0;
61200              FILESPEC;
61300              if not WRONGFILE
61400                then OFILE(FILENAME, OCTFLAG)
61500              else WRITELN('??', BELL)
61600
61700                                  10
61800
61900
62000
62100            end;
62200          'R', 'r':
62300            begin
62400              SETMODE(OCTFLAG, ZHSRMODE);
62500              DEVICE := '      '; FILENAME := '          ';
62600              LEFTPPN := 0; RIGHTPPN := 0; PROTECTION := 0;
62700              FILESPEC;
62800              if not WRONGFILE
62900                then IFILE(FILENAME, OCTFLAG)
63000              else WRITELN('??', BELL)
63100            end;
63200          'S', 's': SETBRKFLAG(ZSINGLE, true);
63300          'T', 't':
63400            begin
63500              LAPS := REALTIME - TIMER;
63600              TIMER := REALTIME;
63700              WRITELN('    ', LAPS div 10, ' micro seconds')
63800            end;
63900          'W', 'w':
64000            begin
64100              CONDITION := true;
64200              for I in 0 to ((1 * MEMSIZE) div 2)
64300                  do
64400              begin
64500                MREADWORD(I, REQWORD);
64600                if LOGAND(CCI, SEARCHMASK) = LOGAND(REQWORD,
64700                  SEARCHMASK)
64800                  then
64900                begin
65000                  WRITELN(O: 6, I, O: 10, REQWORD);
65100                  CONDITION := false;
65200                  exit
65300                end
65400              end;
65500              if CONDITION
65600                then WRITELN( % BELL,\'Search Fails')
65700            end;
65800          's':
65900            begin
66000              CONDITION := true;
```

```
66100            if OCTFLAG
66200              then
66300            begin
66400              GETOCT(CH, OCTFLAG, OCT);
66500              OCT := OCT mod 8;
66600              if (CH = 'A') or (CH = 'a')
66700                then
66800              begin
66900                SETBREAK(ZADRBREAK, OCT, 0);
67000                for I in 0 to MAXBREAK
67100
67200                                    11
67300
67400
67500
67600                                  do
67700                  if CONDITION and (ADRBREAK[I] <> 0)
67800                    then CONDITION := false;
67900                if CONDITION
68000                  then SETBRKFLAG(ZADRFLAG, false);
68100              end
68200              else
68300                if (CH = 'B') or (CH = 'b')
68400                  then
68500                begin
68600                  SETBREAK(ZINSBREAK, OCT, 0);
68700                  for I in 0 to MAXBREAK
68800                    do
68900                  if CONDITION and (INSBREAK[I] <> 0)
69000                    then CONDITION := false;
69100                  if (CONDITION)
69200                    then SETBRKFLAG(ZINSFLAG, false);
69300                end
69400                else WRITELN( % BELL,\
69500
69600    ' ? ILLEGAL CHARA-SPECIAL CHAR SHOULD BE A OR B '
69700                                  )
69800            end
69900            else
70000            begin
70100              GETOCT(CH, OCTFLAG, OCT);
70200              OCT := OCT mod 8;
70300              if (CH = 'A') or (CH = 'a')
70400                then WRITELN(O: 6, ADRBREAK[OCT])
70500              else
70600                if (CH = 'B') or (CH = 'b')
70700                  then WRITELN(O: 6, INSBREAK[OCT])
70800                else WRITELN( % BELL,\
70900
71000    ' ? ILLEGAL CHARA-SPECIAL CHAR SHOULD BE A OR B'
71100                                  )
71200            end
71300          end;
71400        '/', '\';
71500          begin
71600            if not OCTFLAG
71700              then OCT := OLDADDRESS;
71800            DEBUG(CH, OCT);
71900            SETTRAP(false, TRAP);
72000            SETTRAP(false, BUSTRAP)
```

```
72100                end;
72200              otherwise:
72300                begin
72400                  if CH = CR
72500                    then CH := LF;
72600
72700                                                    12
```

```
63400                   if (CH <> CR) and (CH <> LF) and (CH <> CHR(0))
63500                      then WRITELN( % BELL,\'ILLEGAL SPECIAL CHARACTER ')
63600                end
63700            end % CASE \
63800         end % LOOP \
63900      end DIALOG.
```

```
**END** User KRISHNA [70,103] Job DIALOG Seq. 1647 Date 10-May-82 18:06:24
```



```
**END** User KRISHNA [70,103] Job DIALOG Seq. 1647 Date 10-May-82 18:06:24
```



```
* * * L P T S P L   R u n   L o g * * *

18:05:08 LPDAT  [LPTLSJ  LPTSPL version 102(2263) running on LPT101, 10-May-8
18:05:08 LPDAT  [LPTSJS  Starting Job DIALOG, Seq #1647, request created at 1
18:05:18 LPMSG  [LPTSTF  Starting File DSKC:QVL101.LPT<077>[3,3](DIALOG)]
18:06:22 LPMSG  [LPTFPF  Finished Printing File DSKC:QVL101.LPT<077>[3,3](DIA
18:06:24 LPSUM  Spooler runtime 3 Seconds, 14 KCS, 51 disk reads, 13 pages pr
```



```
**END** User KRISHNA [70,103] Job DIALOG Seq. 1647 Date 10-May-82 18:06:24

**END** User KRISHNA [70,103] Job DIALOG Seq. 1647 Date 10-May-82 18:06:24

**END** User KRISHNA [70,103] Job DIALOG Seq. 1647 Date 10-May-82 18:06:24

**END** User KRISHNA [70,103] Job DIALOG Seq. 1647 Date 10-May-82 18:06:24

**END** User KRISHNA [70,103] Job DIALOG Seq. 1647 Date 10-May-82 18:06:24

**END** User KRISHNA [70,103] Job DIALOG Seq. 1647 Date 10-May-82 18:06:24

**END** User KRISHNA [70,103] Job DIALOG Seq. 1647 Date 10-May-82 18:06:24

**END** User KRISHNA [70,103] Job DIALOG Seq. 1647 Date 10-May-82 18:06:24

**END** User KRISHNA [70,103] Job DIALOG Seq. 1647 Date 10-May-82 18:06:24

**END** User KRISHNA [70,103] Job DIALOG Seq. 1647 Date 10-May-82 18:06:24

**END** User KRISHNA [70,103] Job DIALOG Seq. 1647 Date 10-May-82 18:06:24
```

START User KRISHNA [70,103] Job UTILITY Seq. 1603 Date 10-May-82 17:32:23

START User KRISHNA [70,103] Job UTILITY Seq. 1603 Date 10-May-82 17:32:23

```
K   K  RRRR     III     SSSS  H   H  N   N   AAA
K  K   R   R     I     S      H   H  NN  N  A   A
K K    R   R     I     S      H   H  N N N  A   A
KK     RRRR      I      SSS   HHHHH  N N N  A   A
K K    R R       I         S  H   H  N  NN  AAAAA
K  K   R  R      I         S  H   H  N   N  A   A
K   K  R   R   III    SSSS   H   H  N   N  A   A
```

START User KRISHNA [70,103] Job UTILITY Seq. 1603 Date 10-May-82 17:32:23

START User KRISHNA [70,103] Job UTILITY Seq. 1603 Date 10-May-82 17:32:23

START User KRISHNA [70,103] Job UTILITY Seq. 1603 Date 10-May-82 17:32:23

START User KRISHNA [70,103] Job UTILITY Seq. 1603 Date 10-May-82 17:32:23

START User KRISHNA [70,103] Job UTILITY Seq. 1603 Date 10-May-82 17:32:23

START User KRISHNA [70,103] Job UTILITY Seq. 1603 Date 10-May-82 17:32:23

START User KRISHNA [70,103] Job UTILITY Seq. 1603 Date 10-May-82 17:32:23

```
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000001111111112222222223333333334444444445555555556666666667777777
1234567890123456789012345678901234567890123456789012345678901234567890123456789012345
```

```
module UTILITY options(VERSION = ASIM, DEFAULTCXT)
  include IOSYS;
  type
    VIEWTYPE = (DECWORD, TDCWORD, BYTE, BITS, BYTESELECT,
                PRIORITY);
    WORDTYPE =
      packed record
        case VIEWTYPE of
          DECWORD: (FULLWORD: integer);
          TDCWORD: (DUMMY1TDC: 0 .. 17B;
          LEFTWORD, RIGHTWORD: 0 .. 177777B);
          BYTE: (DUMMYBYTE: 0 .. 17B;
          BYTE3, BYTE2, BYTE1, BYTE0: 0 .. 377B);
          BITS: (BITVAL: packed array [1 .. 36] of boolean);
          BYTESELECT: (IGNORE7: 0 .. 777777777777B;
          BYTESELECTOR: 0 .. 3B);
          PRIORITY: (IGNORE8: 0 .. 777777777B;
          PRTYMSK: 0 .. 7; IGNORE9: 0 .. 37B)
      end WORDTYPE;
  type export
    COMPSELECT = (BYT, WORDS);
    TYPEWORD = (TOTALWORD, LEFT, RIGHT);
    TYPEBYTE = (BYT0, BYT1, BYT2, BYT3);
  function export BELL return char;
    begin
      return CHR(07)
    end BELL;
  function export BIT(WORD,BITNUM:integer) return boolean;
    type
      T = packed array [1 .. 36] of boolean;
    var
      B: T;
    begin
      B := T: WORD;
      if (1 <= BITNUM) and (BITNUM <= 36)
        then return B[BITNUM]
      else
      begin
        WRITELN(BELL,
        'ARRAY INDEX EXCEEDED 36 IN FUNCTION-BIT');
        return
      end
    end BIT;
  procedure export SETBIT(BITNUM:integer; NEWVALUE:boolean;
                          var LOCATION:integer);
    var
      R1: WORDTYPE(DECWORD);
      R8: WORDTYPE(BITS);
    begin
      R1.FULLWORD := LOCATION;
      R8 := WORDTYPE(BITS); R1;
      R8.BITVAL[BITNUM] := NEWVALUE;
      R1 := WORDTYPE(DECWORD): R8;
      LOCATION := R1.FULLWORD
    end SETBIT;
  procedure export GETBYTE(FROMWORD:integer;
                           BYTENUM:TYPEBYTE;
                           var WANTEDWORD:integer);
    var
```

```
    R1: WORDTYPE(DECWORD);
    R3: WORDTYPE(BYTE);
  begin
    R1.FULLWORD := FROMWORD;
    R3 := WORDTYPE(BYTE): R1;
    case BYTENUM of
      BYT0: WANTEDWORD := R3.BYTE0;

      BYT1: WANTEDWORD := R3.BYTE1;
      BYT2: WANTEDWORD := R3.BYTE2;
      BYT3: WANTEDWORD := R3.BYTE3
    end
  end GETBYTE;
Procedure export SETBYTE(BYTENUM:TYPEBYTE;
                         NEWBYTE:integer;
                         var LOCATION:integer);
  var
    R1: WORDTYPE(DECWORD);
    R3: WORDTYPE(BYTE);
  begin
    R1.FULLWORD := LOCATION;
    R3 := WORDTYPE(BYTE): R1;
    case BYTENUM of
      BYT0: R3.BYTE0 := NEWBYTE;
      BYT1: R3.BYTE1 := NEWBYTE;
      BYT2: R3.BYTE2 := NEWBYTE;
      BYT3: R3.BYTE3 := NEWBYTE
    end;
    R1 := WORDTYPE(DECWORD): R3;
    LOCATION := R1.FULLWORD
  end SETBYTE;
Procedure export GETWORD(FROMWORD:integer; WORD:TYPEWORD;
                         var WANTED:integer);
  var
    R1: WORDTYPE(DECWORD);
    R2: WORDTYPE(TDCWORD);
  begin
    R1.FULLWORD := FROMWORD;
    R2 := WORDTYPE(TDCWORD): R1;
    case WORD of
      LEFT: WANTED := R2.LEFTWORD;
      RIGHT: WANTED := R2.RIGHTWORD
    end
  end GETWORD;
Procedure export SETWORD(WORDNUM:TYPEWORD;
                         NEWWORD:integer;
                         var LOCATION:integer);
  var
    R1: WORDTYPE(DECWORD);
    R2: WORDTYPE(TDCWORD);
  begin
    if WORDNUM = TOTALWORD
    then LOCATION := NEWWORD
    else
    begin
      R1.FULLWORD := LOCATION;
      R2 := WORDTYPE(TDCWORD): R1;
      case WORDNUM of
```

```
        LEFT: R2.LEFTWORD := NEWWORD;
        RIGHT: R2.RIGHTWORD := NEWWORD
      end;
      R1 := WORDTYPE(DECWORD): R2;
      LOCATION := R1.FULLWORD
  end
end SETWORD;
Procedure export SETPRTY(NEWVALUE:integer;
                         var LOCATION:integer);
  var
    R1: WORDTYPE(DECWORD);
    R10: WORDTYPE(PRIORITY);
  begin
    R1.FULLWORD := LOCATION;
    R10 := WORDTYPE(PRIORITY): R1;
    R10.PRTYMSK := NEWVALUE;
    WRITELN('r10.prtymsk=', R10.PRTYMSK);
    R1 := WORDTYPE(DECWORD): R10;
    LOCATION := R1.FULLWORD;
    WRITELN('location=', LOCATION)
  end SETPRTY;
Procedure export COMPLEMENT(WORD:integer;
                            SELECTOR:COMPSELECT;
                            var COMP:integer);
  var
    R3: WORDTYPE(BYTE);
    R2: WORDTYPE(TDCWORD);
    R1: WORDTYPE(DECWORD);
    I: integer;
    R8: WORDTYPE(BITS);
  begin
    COMP := 0;
    I := 36;
    if (SELECTOR = BYT) or (SELECTOR = WORDS) then
    begin
      R1.FULLWORD := WORD;
      R8 := WORDTYPE(BITS): R1;
      while not (R8.BITVAL[I])
        do I := I - 1;
      I := I - 1;
      while I <> 0 do
      begin
        R8.BITVAL[I] := not (R8.BITVAL[I]);
        I := I - 1
      end;
      case SELECTOR of
        BYT:
          begin
            R3 := WORDTYPE(BYTE): R8;
            COMP := R3.BYTE0
```

```
            end;
          WORDS:
            begin
              R2 := WORDTYPE(DECWORD): R8;
              COMP := R2.RIGHTWORD
            end
        end
      end
      else WRITELN(BELL,
'error in s
elector:selector=-BYT-  for byte and -WORDS- forword');
    end COMPLEMENT;
  procedure export EXTEND(WORD:integer; SELECTOR:COMPSELECT;
                          var EXTENDED:integer);
    var
      VALBIT, I, J: integer;
      R1: WORDTYPE(DECWORD);
      R2: WORDTYPE(BITS);
    begin
      EXTENDED := 0;
      if (SELECTOR = BYT) or (SELECTOR = WORDS)
        then
      begin
        if SELECTOR = BYT
          then J := 21
        else J := 29;
        R1.FULLWORD := WORD;
        R2 := WORDTYPE(BITS): R1;
        for I in 1 to J
          do R2.BITVAL[I] := R2.BITVAL[J];
        R1 := WORDTYPE(DECWORD): R2;
        EXTENDED := R1.FULLWORD
      end
      else WRITELN(BELL,

'error in selector :selector=BYT for byte =WORDS for word')
    end EXTEND;
  procedure export LSHIFT(INPUT,POSITIONS:integer;
                          var SHIFTED:integer);
    var
      R1: WORDTYPE(DECWORD);
      R8: WORDTYPE(BITS);
      I, J: integer;
    begin
      R1.FULLWORD := INPUT;
      R8 := WORDTYPE(BITS): R1;
      for I in 1 to POSITIONS
        do
      begin
```

```
      for J in 2 to 35
        do R8.BITVAL[J] := R8.BITVAL[J + 1];
      R8.BITVAL[36] := false
    end;
    R1 := WORDTYPE(DECWORD): R8;
    SHIFTED := R1.FULLWORD
  end LSHIFT;
procedure export RSHIFT(INPUT,POSITIONS:integer;
                        var SHIFTED:integer);
  var
    I, J: integer;
    R1: WORDTYPE(DECWORD);
    R8: WORDTYPE(BITS);
  begin
    R1.FULLWORD := INPUT;
    R8 := WORDTYPE(BITS): R1;
    for I in 1 to POSITIONS
      do
    begin
      for J in 36 downto 3
        do R8.BITVAL[J] := R8.BITVAL[J - 1];
      R8.BITVAL[2] := R8.BITVAL[1]
    end;
    R1 := WORDTYPE(DECWORD): R8;
    SHIFTED := R1.FULLWORD
  end RSHIFT;
function export LOGAND(WORD,MASK:integer) return integer;
  var
    TEMP: integer;
    R1: WORDTYPE(DECWORD);
    R8: WORDTYPE(BITS);
  begin
    TEMP := 0;
    R1.FULLWORD := TEMP;
    R8 := WORDTYPE(BITS): R1;
    for I in 1 to 36
      do R8.BITVAL[I] := BIT(WORD, I) and BIT(MASK, I);
    R1 := WORDTYPE(DECWORD): R8;
    TEMP := R1.FULLWORD;
    return TEMP
  end LOGAND;
function export LOGOR(WORD,MASK:integer) return integer;
  var
    TEMP: integer;
    R1: WORDTYPE(DECWORD);
    R8: WORDTYPE(BITS);
  begin
    TEMP := 0;
    R1.FULLWORD := TEMP;
    R8 := WORDTYPE(BITS): R1;
```

```
22800           do R8,BITVAL[I] := BIT(WORD, I) and BIT(MASK, I);
22900         R1 := WORDTYPE(DECWORD): R8;
23000         TEMP := R1.FULLWORD;
23100         return TEMP
23200       end LOGAND;
23300     function export LOGOR(WORD,MASK:integer) return integer;
23400       var
23500         TEMP: integer;
23600         R1: WORDTYPE(DECWORD);
23700         R8: WORDTYPE(BITS);
23800       begin
23900         TEMP := 0;
24000         R1.FULLWORD := TEMP;
24100         R8 := WORDTYPE(BITS): R1;
24200         for I in 1 to 36
24300           do R8,BITVAL[I] := BIT(WORD, I) or BIT(MASK, I);
24400         R1 := WORDTYPE(DECWORD): R8;
24500         TEMP := R1.FULLWORD;
24600         return TEMP
24700       end LOGOR;
24800     function export NOTT(WORD:integer) return integer;
24900       var
25000         I: integer;
25100         R1: WORDTYPE(DECWORD);
25200         R8: WORDTYPE(BITS);
25300       begin
25400         R1.FULLWORD := WORD;
25500         R8 := WORDTYPE(BITS): R1;
25600         for I in 1 to 36
25700           do R8,BITVAL[I] := not (R8.BITVAL[I]);
25800         R1 := WORDTYPE(DECWORD): R8;
25900         return R1.FULLWORD
26000       end NOTT;
26100     begin % utilty \
26200     end UTILTY.
```

```
**END** User KRISHNA [70,103]  Job  UTILTY Seq. 1603 Date 10-May-82 17:32:50

**END** User KRISHNA [70,103]  Job  UTILTY Seq. 1603 Date 10-May-82 17:32:50

                        * * * L P T S P L   R u n   L o g * * *

17:32:17 LPDAT  [LPTLSJ  LPTSPL version 102(2263) running on LPT101, 10-May-8
17:32:17 LPDAT  [LPTSJS  Starting Job UTILTY, Seq #1603, request created at 1
17:32:23 LPMSG  [LPTSTF  Starting File DSKC:QUZ101.LPT<077>[3,3](UTILTY)]
17:32:50 LPMSG  [LPTFPF  Finished Printing File DSKC:QUZ101.LPT<077>[3,3](UTI
17:32:50 LPSUM  Spooler runtime 0 seconds, 6 KCS, 19 disk reads, 6 pages prin

**END** User KRISHNA [70,103]  Job  UTILTY Seq. 1603 Date 10-May-82 17:32:50

**END** User KRISHNA [70,103]  Job  UTILTY Seq. 1603 Date 10-May-82 17:32:50

**END** User KRISHNA [70,103]  Job  UTILTY Seq. 1603 Date 10-May-82 17:32:50

**END** User KRISHNA [70,103]  Job  UTILTY Seq. 1603 Date 10-May-82 17:32:50

**END** User KRISHNA [70,103]  Job  UTILTY Seq. 1603 Date 10-May-82 17:32:50

**END** User KRISHNA [70,103]  Job  UTILTY Seq. 1603 Date 10-May-82 17:32:50

**END** User KRISHNA [70,103]  Job  UTILTY Seq. 1603 Date 10-May-82 17:32:50

**END** User KRISHNA [70,103]  Job  UTILTY Seq. 1603 Date 10-May-82 17:32:50

**END** User KRISHNA [70,103]  Job  UTILTY Seq. 1603 Date 10-May-82 17:32:50

**END** User KRISHNA [70,103]  Job  UTILTY Seq. 1603 Date 10-May-82 17:32:50

**END** User KRISHNA [70,103]  Job  UTILTY Seq. 1603 Date 10-May-82 17:32:50
```

START User KRISHNA [70,105] Job GLOBAL Seq. 2039 Date 18-May-82 11:56:44

START User KRISHNA [70,105] Job GLOBAL Seq. 2039 Date 18-May-82 11:56:44

```
K   K  RRRR    III    SSSS  H   H  N   N  AAA
K   K  R   R    I    S      H   H  N   N  A   A
K  K   R   R    I    S      H   H  NN  N  A   A
KKK    RRRR     I     SSS   HHHHH  N N N  A   A
K  K   R R      I        S  H   H  N  NN  AAAAA
K   K  R  R     I        S  H   H  N   N  A   A
K   K  R   R   III   SSSS   H   H  N   N  A   A
```

START User KRISHNA [70,105] Job GLOBAL Seq. 2039 Date 18-May-82 11:56:44

START User KRISHNA [70,105] Job GLOBAL Seq. 2039 Date 18-May-82 11:56:44

START User KRISHNA [70,105] Job GLOBAL Seq. 2039 Date 18-May-82 11:56:44

START User KRISHNA [70,105] Job GLOBAL Seq. 2039 Date 18-May-82 11:56:44

START User KRISHNA [70,105] Job GLOBAL Seq. 2039 Date 18-May-82 11:56:44

START User KRISHNA [70,105] Job GLOBAL Seq. 2039 Date 18-May-82 11:56:44

START User KRISHNA [70,105] Job GLOBAL Seq. 2039 Date 18-May-82 11:56:44

```
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000001111111111222222222233333333334444444455555555556666666666777777777
1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678
```

```
00100    module GLOBAL options(VERSION = NSIM, DEFAULTCXT)
00200      include IOSYS;
00300      const export
00400        DEBUGGING = false;
00500      var export
00600        TRAPFLAG, STRAPFLAG, BTRAPFLAG, RTRAPFLAG, BUSTRAPFLAG,
00700        ILLTRAPFLAG,
00800        HALTFLAG, BYTEFLAG, wait,
00900        REGMODE, RUNMODE, SPMODE, HSRMODE, HSPMODE: boolean;
01000        REALTIME: integer;
01100      type export
01200        TRAPTYPE = (TRAP, STRAP, BTRAP, RTRAP, ILLTRAP, BUSTRAP)
01300        ,
01400        MODETYPE = (ZREGMODE, ZRUNMODE, ZSPMODE, ZHSRMODE,
01500                    ZHSPMODE);
01600        FLAGTYPE = (ZHALTFLAG, ZBYTEFLAG, ZWAIT);
01700      procedure export SETTRAP(NEWVALUE:boolean;
01800                               LOCATION:TRAPTYPE);
01900        begin
02000          case LOCATION of
02100            TRAP: TRAPFLAG := NEWVALUE;
02200            STRAP: STRAPFLAG := NEWVALUE;
02300            BTRAP: BTRAPFLAG := NEWVALUE;
02400            RTRAP: RTRAPFLAG := NEWVALUE;
02500            ILLTRAP: ILLTRAPFLAG := NEWVALUE;
02600            BUSTRAP: BUSTRAPFLAG := NEWVALUE
02700          end
02800        end SETTRAP;
02900      procedure export SETMODE(NEWVALUE:boolean; MODE:MODETYPE);
03000        begin
03100          case MODE of
03200            ZREGMODE: REGMODE := NEWVALUE;
03300            ZRUNMODE: RUNMODE := NEWVALUE;
03400            ZSPMODE: SPMODE := NEWVALUE;
03500            ZHSRMODE: HSRMODE := NEWVALUE;
03600            ZHSPMODE: HSPMODE := NEWVALUE
03700          end
03800        end SETMODE;
03900      procedure export SETFLAG(NEWVALUE:boolean; FLAG:FLAGTYPE);
04000        begin
04100          case FLAG of
04200            ZHALTFLAG: HALTFLAG := NEWVALUE;
04300            ZBYTEFLAG: BYTEFLAG := NEWVALUE;
04400            ZWAIT: wait := NEWVALUE
04500          end
04600        end SETFLAG;
04700      procedure export SETTIME(NEWVALUE:integer);
04800        begin
04900          REALTIME := NEWVALUE
05000        end SETTIME;
05100      begin %global\
05200        REALTIME := 0; SPMODE := false; BYTEFLAG := false;
05300        wait := false;
05400        for I in TRAP to BUSTRAP
05500          do SETTRAP(false, I);
05600        HALTFLAG := false
05700      end GLOBAL.
```

```
**END** User KRISHNA [70,105]  Job  GLOBAL Seq. 2039 Date 18-May-82 11:56:59

**END** User KRISHNA [70,105]  Job  GLOBAL Seq. 2039 Date 18-May-82 11:56:59

                         * * * L P T S P L  R u n  L o g * * *

11:56:43 LPDAT  [LPTLSJ  LPTSPL version 102(2263) running on LPT101, 18-May-8
11:56:43 LPDAT  [LPTSJS  Starting Job GLOBAL, Seq #2039, request created at 1
11:56:44 LPMSG  [LPTSTF  Starting File DSKC:Q2W101.LPT<077>[3,3](GLOBAL)]
11:56:50 LPMSG  [LPTFPF  Finished Printing File DSKC:Q2W101.LPT<077>[3,3](GLO
11:56:59 LPSUM  Spooler runtime 0 Seconds, 4 KCS, 10 disk reads, 1 pages prin

**END** User KRISHNA [70,105]  Job  GLOBAL Seq. 2039 Date 18-May-82 11:56:59

**END** User KRISHNA [70,105]  Job  GLOBAL Seq. 2039 Date 18-May-82 11:56:59

**END** User KRISHNA [70,105]  Job  GLOBAL Seq. 2039 Date 18-May-82 11:56:59

**END** User KRISHNA [70,105]  Job  GLOBAL Seq. 2039 Date 18-May-82 11:56:59

**END** User KRISHNA [70,105]  Job  GLOBAL Seq. 2039 Date 18-May-82 11:56:59

**END** User KRISHNA [70,105]  Job  GLOBAL Seq. 2039 Date 18-May-82 11:56:59

**END** User KRISHNA [70,105]  Job  GLOBAL Seq. 2039 Date 18-May-82 11:56:59

**END** User KRISHNA [70,105]  Job  GLOBAL Seq. 2039 Date 18-May-82 11:56:59

**END** User KRISHNA [70,105]  Job  GLOBAL Seq. 2039 Date 18-May-82 11:56:59

**END** User KRISHNA [70,105]  Job  GLOBAL Seq. 2039 Date 18-May-82 11:56:59

**END** User KRISHNA [70,105]  Job  GLOBAL Seq. 2039 Date 18-May-82 11:56:59
```

START User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:31

START User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:31

```
K   K  RRRR    III   SSSS  H   H  N   N   AAA
K  K   R   R    I   S      H   H  NN  N  A   A
K K    R   R    I   S      H   H  N N N  A   A
KK     RRRR     I    SSS   HHHHH  N  NN  A   A
K K    R R      I       S  H   H  N   N  AAAAA
K  K   R  R     I       S  H   H  N   N  A   A
K   K  R   R   III  SSSS   H   H  N   N  A   A
```

START User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:31

START User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:31

START User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:31

START User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:31

START User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:31

START User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:31

START User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:31

START User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:31

START User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:31

```
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000011111111112222222222333333333344444444445555555555666666666677777777
12345678901234567890123456789012345678901234567890123456789012345678901234567
```

```
00100  module GLOBAL options(VERSION = NSIM, DEFAULTCXT)
00200    include IOSYS;
00300    const export
00400      DEBUGGING = false;
00500    var export
00600      TRAPFLAG, STRAPFLAG, BTRAPFLAG, RTRAPFLAG, BUSTRAPFLAG,
00700      ILLTRAPFLAG,
00800      HALTFLAG, BYTEFLAG, wait,
00900      REGMODE, RUNMODE, SPMODE, HSRMODE, HSPMODE: boolean;
01000      REALTIME: integer;
01100    type export
01200      TRAPTYPE = (TRAP, STRAP, BTRAP, RTRAP, ILLTRAP, BUSTRAP)
01300      ,
01400      MODETYPE = (ZREGMODE, ZRUNMODE, ZSPMODE, ZHSRMODE,
01500                  ZHSPMODE);
01600      FLAGTYPE = (ZHALTFLAG, ZBYTEFLAG, ZWAIT);
01700    procedure export SETTRAP(NEWVALUE:boolean;
01800                             LOCATION:TRAPTYPE);
01900      begin
02000        case LOCATION of
02100          TRAP: TRAPFLAG := NEWVALUE;
02200          STRAP: STRAPFLAG := NEWVALUE;
02300          BTRAP: BTRAPFLAG := NEWVALUE;
02400          RTRAP: RTRAPFLAG := NEWVALUE;
02500          ILLTRAP: ILLTRAPFLAG := NEWVALUE;
02600          BUSTRAP: BUSTRAPFLAG := NEWVALUE
02700        end
02800      end SETTRAP;
02900    procedure export SETMODE(NEWVALUE:boolean; MODE:MODETYPE);
03000      begin
03100        case MODE of
03200          ZREGMODE: REGMODE := NEWVALUE;
03300          ZRUNMODE: RUNMODE := NEWVALUE;
03400          ZSPMODE: SPMODE := NEWVALUE;
03500          ZHSRMODE: HSRMODE := NEWVALUE;
03600          ZHSPMODE: HSPMODE := NEWVALUE
03700        end
03800      end SETMODE;
03900    procedure export SETFLAG(NEWVALUE:boolean; FLAG:FLAGTYPE);
04000      begin
04100        case FLAG of
04200          ZHALTFLAG: HALTFLAG := NEWVALUE;
04300          ZBYTEFLAG: BYTEFLAG := NEWVALUE;
04400          ZWAIT: wait := NEWVALUE
04500        end
04600      end SETFLAG;
04700    procedure export SETTIME(NEWVALUE:integer);
04800      begin
04900        REALTIME := NEWVALUE
05000      end SETTIME;
05100    begin %global\
05200      REALTIME := 0; SPMODE := false; BYTEFLAG := false;
05300      wait := false;
05400      for I in TRAP to BUSTRAP
05500        do SETTRAP(false, I);
05600      HALTFLAG := false
05700    end GLOBAL.
```

```
**END** User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:35

**END** User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:35

                    * * * L P T S P L  R u n  L o g * * *

11:56:30 LPDAT  [LPTLSJ  LPTSPL version 102(2263) running on LPT101, 18-May-8
11:56:30 LPDAT  [LPTSJS  Starting Job GLOBAL, Seq #2038, request created at 1
11:56:31 LPMSG  [LPTSTF  Starting File DSKC:Q2V101.LPT<077>[3,3](GLOBAL)]
11:56:34 LPMSG  [LPTFPF  Finished Printing File DSKC:Q2V101.LPT<077>[3,3](GLO
11:56:35 LPSUM  Spooler runtime 0 Seconds, 4 KCS, 10 disk reads, 1 pages prin

**END** User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:35

**END** User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:35

**END** User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:35

**END** User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:35

**END** User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:35

**END** User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:35

**END** User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:35

**END** User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:35

**END** User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:35

**END** User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:35

**END** User KRISHNA [70,105] Job GLOBAL Seq. 2038 Date 18-May-82 11:56:35
```

START User KRISHNA [70,105] Job FILEOP Seq. 2212 Date 18-May-82 12:54:44

START User KRISHNA [70,105] Job FILEOP Seq. 2212 Date 18-May-82 12:54:44

```
K    K  RRRR    III    SSSS  H    H  N    N   AAA
K   K   R   R    I    S      H    H  N    N  A   A
K  K    R   R    I    S      H    H  NN   N  A   A
KKK     RRRR     I     SSS   HHHHHH  N N  N  A   A
K  K    R R      I        S  H    H  N  NN   AAAAA
K   K   R  R     I        S  H    H  N    N  A   A
K    K  R   R   III   SSSS   H    H  N    N  A   A
```

START User KRISHNA [70,105] Job FILEOP Seq. 2212 Date 18-May-82 12:54:44

START User KRISHNA [70,105] Job FILEOP Seq. 2212 Date 18-May-82 12:54:44

START User KRISHNA [70,105] Job FILEOP Seq. 2212 Date 18-May-82 12:54:44

START User KRISHNA [70,105] Job FILEOP Seq. 2212 Date 18-May-82 12:54:44

START User KRISHNA [70,105] Job FILEOP Seq. 2212 Date 18-May-82 12:54:44

START User KRISHNA [70,105] Job FILEOP Seq. 2212 Date 18-May-82 12:54:44

START User KRISHNA [70,105] Job FILEOP Seq. 2212 Date 18-May-82 12:54:44

START User KRISHNA [70,105] Job FILEOP Seq. 2212 Date 18-May-82 12:54:44

START User KRISHNA [70,105] Job FILEOP Seq. 2212 Date 18-May-82 12:54:44

```
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000011111111112222222222333333333344444444445555555555666666666677777777
12345678901234567890123456789012345678901234567890123456789012345678901234567
```

```
00100
00200   module FILEOP options(VERSION = NSIM, DEFAULTCXT);
00300     include MEMORY, IOSYS;
00400     type
00500       VIEWTYPE = (DECWORD, TDCBYTES, BYTE, BITS);
00600       WORDTYPE =
00700         packed record
00800           case VIEWTYPE of
00900             DECWORD: (FULLWORD: integer);
01000             TDCBYTES: (JUNK1: 0 .. 3B;
01100             TDCBYTE1, TDCBYTE0: 0 .. 377B;
01200                        JUNK2: 0 .. 3B;
01300             TDCBYTE3, TDCBYTE2: 0 .. 377B);
01400             BYTE: (DUMMYBYTE: 0 .. 17B;
01500                    BYTE3, BYTE2, BYTE1, BYTE0: 0 .. 377B);
01600             BITS: (BITVAL: packed array [1 .. 36] of boolean)
01700         end WORDTYPE;
01800     var
01900       HSRBIN, HSPBIN: BINARYFILE;
02000       HSRTEXT, HSPTEXT: TEXTFILE;
02100     procedure export ICLOSE;
02200       begin
02300         if not (BIT(HSRSTATUS, 21))
02400           then IORITE(HSR, 100000B)
02500       end ICLOSE;
02600     procedure export OCLOSE;
02700       begin
02800         if not (BIT(HSPSTATUS, 21))
02900           then IORITE(HSP, 100000B)
03000       end OCLOSE;
03100     procedure export IFILE(FILENAME:string(9);
03200       OCTFLAG:boolean);
03300       begin
03400         ICLOSE;
03500         if OCTFLAG
03600           then RESET(HSRBIN, FILENAME)
03700           else RESET(HSRTEXT, FILENAME);
03800         IORITE(HSR, 0); WRITELN('HSRSTATUS=', HSRSTATUS)
03900       end IFILE;
04000     procedure export OFILE(FILENAME:string(9);
04100       OCTFLAG:boolean);
04200       begin
04300         OCLOSE;
04400         if OCTFLAG
04500           then REWRITE(HSPBIN, FILENAME)
04600           else REWRITE(HSPTEXT, FILENAME);
04700         IORITE(HSP, 0); WRITELN('HSPSTATUS=', 0: 6, HSPSTATUS)
04800       end OFILE;
04900     procedure export LOADFILE(OCTFLAG:boolean;
05000                               FILENAME:string(9));
05100       %*************************************************
05200       *********
05300       *\
05400       % loads an absolutes binary file into simulated core \
05500       %TDCbytes are in the format(JUNK1: 0 .. 3B; TDCBYTE1, TD
05600       CBYTE0: 0
05700         .. 377B; JUNK2: 0 .. 3B; TDCBYTE3, TDCBYTE2: 0 ..
05800         377B);\
05900       var
06000         BYTECOUNT, BYTEVALUE, I, CKSUM, ADDRESS, COMPBYTE,
```

```
06100          COUNT,
06200          WANTEDBYTE: integer;
06300          R1: WORDTYPE(DECWORD);
06400          R3: WORDTYPE(BYTE);
06500          R6: WORDTYPE(TDCBYTES);
06600        procedure GETTDCBYTE;
06700          %=========================================\
06800
06900
07000
07100          %this procedure returns the 'next' TDCbyte each time i
07200          t is calle
07300           d\
07400          var
07500            R1: WORDTYPE(DECWORD);
07600          begin
07700            if OCTFLAG
07800              then
07900            begin
08000              if not EOF(HSRBIN)
08100                then BYTEVALUE := GET(HSRBIN);
08200            end
08300            else
08400            begin
08500              BYTEVALUE := 0;
08600              BYTECOUNT := (BYTECOUNT + 1) mod 4;
08700              case BYTECOUNT of
08800                0:
08900                  begin
09000                    if not EOF(HSRBIN)
09100                      then R1.FULLWORD := GET(HSRBIN);
09200                    R6 := WORDTYPE(TDCBYTES): R1;
09300                    BYTEVALUE := R6.TDCBYTE0
09400                  end;
09500                1: BYTEVALUE := R6.TDCBYTE1;
09600                2: BYTEVALUE := R6.TDCBYTE2;
09700                3: BYTEVALUE := R6.TDCBYTE3
09800              end
09900            end
10000          end GETTDCBYTE;
10100        begin % loadfile \
10200          BYTECOUNT := - 1;
10300          RESET(HSRBIN, FILENAME);
10400          WRITELN('Loading of ', FILENAME, ' starts ');
10500          loop
10600            BYTEVALUE := 0; COUNT := 0; ADDRESS := 0;
10700            while BYTEVALUE <> 1
10800              do GETTDCBYTE;
10900            CKSUM := 1;
11000            GETTDCBYTE;
11100            CKSUM := CKSUM + BYTEVALUE;
11200            GETTDCBYTE;
11300            CKSUM := CKSUM + BYTEVALUE;
11400            COUNT := BYTEVALUE - 6;
11500            %reason for BYTEVALUE-6 is:first byte contains 1,sec
11600            ond byte i
11700             s just ignored, 3 rd and 4th bytes are used to cal
11800             culate count(the number of bytes) ,5th and 6th byte
11900            s are used
12000             to calculate start address\
```

```
            GETTDCBYTE;
            CKSUM := CKSUM + BYTEVALUE;
            COUNT := COUNT + BYTEVALUE * 256;
            GETTDCBYTE;
            CKSUM := CKSUM + BYTEVALUE;
            ADDRESS := BYTEVALUE;
            GETTDCBYTE;
            CKSUM := CKSUM + BYTEVALUE;
            ADDRESS := BYTEVALUE * 256 + ADDRESS;
            if COUNT < 0
               then
            begin
              WRITELN( % BELL, \' Loader block too small');
              return
            end
            else
            begin
              if COUNT > 0 then
                for I in 1 to COUNT do
                begin
                  GETTDCBYTE;
                  CKSUM := CKSUM + BYTEVALUE;
                  WRITEBYTE(ADDRESS, BYTEVALUE);
                  ADDRESS := ADDRESS + 1
                end;
              GETTDCBYTE;
              COMPLEMENT(CKSUM, BYT, COMPBYTE);
              GETBTE(COMPBYTE, BYTO, WANTEDBYTE);
              if BYTEVALUE <> WANTEDBYTE
                 then
              begin
                WRITELN( % BELL, \
'Checksum error in LOADER block or TRANSFER block ');
                return
              end;
              if COUNT = 0
                then %transfer block\
                begin
                  if not (BIT(ADDRESS, 36))
                     then
                  begin
                    REGSET(ADDRESS, 0);
                    exit
                  end
                  else
                  begin
                    WRITELN( % BELL, \'NO start address');
                    return
                  end
```

```
18100            end
18200          end
18300        end;
18400        % loop \
18500        WRITELN( % BELL, \'Loading Complete')
18600      end LOADFILE;
18700    procedure export DUMP(DBEGIN,DEND:integer;
18800      OCTFLAG:boolean);
18900      %*****************************************\
19000      %this procedure dumps absolute binary file in to the usr
19100      spefied f
19200       ile\
19300      var
19400        I, J, TEMP, REQBYTE, COMPBYTE, TEMPDBEGIN, CKSUM:
19500        integer;
19600      begin
19700        CKSUM := 1; TEMP := 0;
19800        TEMPDBEGIN := DBEGIN;
19900        if BIT(HSPSTATUS, 21)
20000          then
20100        begin
20200          WRITELN( % BELL, \'HSP FILE NOT CREATED');
20300          return
20400        end;
20500        PUT(HSPBIN, 1); PUT(HSPBIN, 0);
20600        if OCTFLAG
20700          then TEMP := (DEND - TEMPDBEGIN + 1) + 6
20800        else TEMP := 6;
20900        GETBTE(TEMP, BYT0, I);
21000        GETBTE(TEMP, BYT1, J);
21100        CKSUM := CKSUM + I + J;
21200        PUT(HSPBIN, I); PUT(HSPBIN, J);
21300        GETBTE(TEMPDBEGIN, BYT0, I);
21400        GETBTE(TEMPDBEGIN, BYT1, J);
21500        CKSUM := CKSUM + I + J;
21600        PUT(HSPBIN, I); PUT(HSPBIN, J);
21700        while TEMPDBEGIN <= DEND
21800              do
21900        begin
22000          MREADBYTE(TEMPDBEGIN, REQBYTE);
22100          CKSUM := CKSUM + REQBYTE;
22200          PUT(HSPBIN, REQBYTE);
22300          TEMPDBEGIN := TEMPDBEGIN + 1
22400        end;
22500        COMPLEMENT(CKSUM, BYT, COMPBYTE);
22600        GETBTE(COMPBYTE, BYT0, I);
22700        PUT(HSPBIN, I);
22800        if not OCTFLAG
22900          then OCLOSE;
23000        WRITELN( % BELL, \'Dumping Complete')
23100
23200                              3
```

```
20100       WRITELN( % BELL, \'Dumping Complete')
20200     end DUMP;
20300   begin % FILEOP \
20400   end FILEOP.
```

```
**END** User KRISHNA [70,105]  Job  FILEOP Seq. 2212 Date 18-May-82 12:55:12

**END** User KRISHNA [70,105]  Job  FILEOP Seq. 2212 Date 18-May-82 12:55:12

                  * * * L P T S P L  R u n  L o g * * *

12:54:44 LPDAT  [LPTLSJ  LPTSPL version 102(2263) running on LPT101, 18-May-8
12:54:44 LPDAT  [LPTSJS  Starting Job FILEOP, Seq #2212, request created at 1
12:54:53 LPMSG  [LPTSTF  Starting File DSKC:FILEOP.DIP<057>[70,105,OVG]]
12:55:12 LPMSG  [LPTFPF  Finished Printing File DSKC:FILEOP.DIP<057>[70,105,O
12:55:12 LPSUM  Spooler runtime 0 Seconds, 5 KCS, 16 disk reads, 4 pages prin

**END** User KRISHNA [70,105]  Job  FILEOP Seq. 2212 Date 18-May-82 12:55:12

**END** User KRISHNA [70,105]  Job  FILEOP Seq. 2212 Date 18-May-82 12:55:12

**END** User KRISHNA [70,105]  Job  FILEOP Seq. 2212 Date 18-May-82 12:55:12

**END** User KRISHNA [70,105]  Job  FILEOP Seq. 2212 Date 18-May-82 12:55:12

**END** User KRISHNA [70,105]  Job  FILEOP Seq. 2212 Date 18-May-82 12:55:12

**END** User KRISHNA [70,105]  Job  FILEOP Seq. 2212 Date 18-May-82 12:55:12

**END** User KRISHNA [70,105]  Job  FILEOP Seq. 2212 Date 18-May-82 12:55:12

**END** User KRISHNA [70,105]  Job  FILEOP Seq. 2212 Date 18-May-82 12:55:12

**END** User KRISHNA [70,105]  Job  FILEOP Seq. 2212 Date 18-May-82 12:55:12

**END** User KRISHNA [70,105]  Job  FILEOP Seq. 2212 Date 18-May-82 12:55:12

**END** User KRISHNA [70,105]  Job  FILEOP Seq. 2212 Date 18-May-82 12:55:12
```

START User KRISHNA [70,105] Job REALS Seq. 2178 Date 18-May-82 12:42:50

START User KRISHNA [70,105] Job REALS Seq. 2178 Date 18-May-82 12:42:50

```
K   K  RRRR    III   SSSS  H   H  N   N  AAA
K   K  R   R    I   S      H   H  N   N  A   A
K  K   R   R    I   S      H   H  NN  N  A   A
KKK    RRRR     I    SSS   HHHHH  N N N  A   A
K  K   R R      I       S  H   H  N  NN  AAAAA
K   K  R  R     I       S  H   H  N   N  A   A
K   K  R   R   III  SSSS   H   H  N   N  A   A
```

START User KRISHNA [70,105] Job REALS Seq. 2178 Date 18-May-82 12:42:50

START User KRISHNA [70,105] Job REALS Seq. 2178 Date 18-May-82 12:42:50

START User KRISHNA [70,105] Job REALS Seq. 2178 Date 18-May-82 12:42:50

START User KRISHNA [70,105] Job REALS Seq. 2178 Date 18-May-82 12:42:50

START User KRISHNA [70,105] Job REALS Seq. 2178 Date 18-May-82 12:42:50

START User KRISHNA [70,105] Job REALS Seq. 2178 Date 18-May-82 12:42:50

START User KRISHNA [70,105] Job REALS Seq. 2178 Date 18-May-82 12:42:50

```
module REALS options(VERSION = NSIM, DEFAULTCXT)
  include MEMORY, IOSYS;
  var export
    CLKSTATUS, KBDSTATUS, HSRSTATUS, HSPSTATUS, TTYSTATUS,
    KBDBUF, HSRBUF, HSPBUF, TTYBUF: integer;
    KBDCHR: char;
  const export
    HSRPRTY = 1; HSPPRTY = 1; KBDPRTY = 1; TTYPRTY = 1;
    CLKPRTY = 6;
    HSRLOCATION = 60B;
    HSPLOCATION = 64B;
    KBDLOCATION = 70B;
    TTYLOCATION = 74B;
    CLKLOCATION = 310B;
    STKLOCATION = 4B;
    TRPLOCATION = 10B;
    BPTLOCATION = 14B;
    EMTLOCATION = 20B;
    IOTLOCATION = 24B;
    RESLOCATION = 30B;
    ILLBUS = 34B;
    IOBUFSIZE = 600;
    HSR = 177660B;
    HSP = 177664B;
    KBD = 177670B;
    TTP = 177674B;
    CLK = 177310B;
    HSR1 = 177661B;
    HSR2 = 177662B;
    HSR3 = 177663B;
    HSP1 = 177665B;
    HSP2 = 177666B;
    HSP3 = 177667B;
    KBD1 = 177671B;
    KBD2 = 177672B;
    KBD3 = 177673B;
    TTP1 = 177675B;
    TTP2 = 177676B;
    TTP3 = 177677B;
    CLK1 = 177311B;
    EQSIZE = 25;
    IQSIZE = 25;
  const
    REG0 = 177740B;
    REG1 = 177742B;
    HSRTIME = 1000;
    HSPTIME = 1000;
    TTYTIME = 1000;
    TRAPTIME = 76;
    KBDTIME = 1000;
    LINEFREQUENCY = 1;
  type
    VIEWTYPE = (DECWORD, IDCWORD, BITS, BYTE, PRIORITY);
    WORDTYPE =
      packed record
        case VIEWTYPE of
          DECWORD: (FULLWORD: integer);
          IDCWORD: (DUMMY1IDC: 0 .. 17B;
                    LEFTWORD, RIGHTWORD: 0 .. 177777B);
```

```
        BITS: (BITVAL: packed array [1 .. 36] of boolean);
        BYTE: (DUMMYBYTE: 0 .. 17B;
               BYTE3, BYTE2, BYTE1, BYTE0: 0 .. 377B);
        PRIORITY: (IGNORE8: 0 .. 177777777B;
        PRTYMASK: 0 .. 7B;
                  IGNORE9: 0 .. 37B)
    end WORDTYPE;


var
  EVENTHEAD, INTHEAD, FREEEVENTPOINTER, FREEINTPOINTER:
  integer;
  QEVENTLINK: array [0 .. EQSIZE] of integer;
  QINTLINK: array [0 .. IQSIZE] of integer;
  CLKFLAG: boolean;
  QEVENT: array [0 .. EQSIZE] of integer;
  QEVENTTIME: array [0 .. EQSIZE] of integer;
  QEVENTPRTY: array [0 .. EQSIZE] of integer;
  QINTPRTY: array [0 .. IQSIZE] of integer;
  QINTVEC: array [0 .. IQSIZE] of integer;
  OLD, INTVEC, WANTEDWORD: integer;
procedure export INITEVENTINTS;
  begin
    for I in 0 to EQSIZE - 1
      do QEVENTLINK[I] := I + 1;
    QEVENTLINK[EQSIZE] :=  - 1;
    for I in 0 to IQSIZE - 1
      do QINTLINK[I] := I + 1;
    QINTLINK[IQSIZE] :=  - 1;
    for I in 0 to EQSIZE do
    begin
      QEVENTTIME[I] := 0;
      QEVENTPRTY[I] := 0;
      QEVENT[I] := 0
    end;
    for I in 0 to IQSIZE do
    begin
      QINTPRTY[I] := 0;
      QINTVEC[I] := 0
    end;
    OLD := 0; INTVEC := 0;
    FREEEVENTPOINTER := 0; FREEINTPOINTER := 0
  end INITEVENTINTS;
procedure export INITSTATUSBUF;
  begin
    KBDSTATUS := 0; TTYSTATUS := 200B; CLKSTATUS := 200B;
    KBDBUF := 0; TTYBUF := 0; HSRBUF := 0; HSPBUF := 0;
    INTHEAD :=  - 1; EVENTHEAD :=  - 1;
    CLKFLAG := true;
    HSRSTATUS := 100000B; HSRBUF := 0;
    HSPSTATUS := 100000B; HSPBUF := 0;
    KBDSTATUS := 0; KBDBUF := 0;
    TTYSTATUS := 200B; TTYBUF := 0;
    CLKSTATUS := 200B
  end INITSTATUSBUF;
procedure export BITSTATUSSET(DIINUM:integer;
                              NEWVALUE:boolean;
                              LOCATION:integer);
  var
```

```
    R1: WORDTYPE(DECWORD);
    R8: WORDTYPE(BITS);
  begin
    if (1 <= BITNUM) and (BITNUM <= 36) then
      case LOCATION of
        KBD: SETBIT(BITNUM, NEWVALUE, KBDSTATUS);
        TTP: SETBIT(BITNUM, NEWVALUE, TTYSTATUS);
        CLK: SETBIT(BITNUM, NEWVALUE, CLKSTATUS);
        HSR: SETBIT(BITNUM, NEWVALUE, HSRSTATUS);
        HSP: SETBIT(BITNUM, NEWVALUE, HSPSTATUS)
      end
    else
    begin
      WRITELN(BELL,
      'index EXCEEDED 36 IN bitstatusset pROCEDURE');
      return
    end
  end BITSTATUSSET;
procedure export MCLREALS;
  begin
    for I in 0 to (EQSIZE - 1)
      do QEVENTLINK[I] := I + 1;
    QEVENTLINK[EQSIZE] :=  - 1;
    for I in 0 to (IQSIZE - 1)
      do QINTLINK[I] := I + 1;
    QINTLINK[IQSIZE] :=  - 1;
    FREEEVENTPOINTER := 0; EVENTHEAD :=  - 1;
    FREEINTPOINTER := 0; INTHEAD :=  - 1;
    KBDSTATUS := 0;
    TTYSTATUS := 200B; CLKSTATUS := 200B;
    if not (BIT(HSRSTATUS, 21))
      then HSRSTATUS := 0
    else HSRSTATUS := 100000B;
    if not (BIT(HSPSTATUS, 21))
      then HSPSTATUS := 0
    else HSPSTATUS := 100000B
  end MCLREALS;
procedure export ENTEREVENT(DEV:integer;
  TIME,PRTY:integer);
  % ENTERS MAXIMUM OF EQSIZE EVENTS IN THE QEVENT LIST \
  var
    CURRENT, NNEW: integer;
  begin
    OLD := 0;
    CURRENT := EVENTHEAD;
    NNEW := 0;
    if (DEV = HSR) and (BIT(HSRSTATUS, 21))
      then
    begin
      WRITELN(BELL, 'HSR NOT READY ');
```

```
    SETMODE(false, ZRUNMODE)
  end;
  if (DEV = HSP) and (BIT(HSPSTATUS, 21))
    then
  begin
    WRITELN(BELL, 'HSP NOT READY ');
    SETMODE(false, ZRUNMODE)
  end;
  while CURRENT >= 0 do
    if (TIME > QEVENTTIME[CURRENT]) or ((TIME =
       QEVENTTIME[CURRENT
       ]) and (PRTY < QEVENTPRTY[CURRENT]))
      then
    begin
      OLD := CURRENT;
      CURRENT := QEVENTLINK[CURRENT]
    end
    else exit;
  if FREEEVENTPOINTER >= 0
    then
  begin
    NNEW := FREEEVENTPOINTER;
    FREEEVENTPOINTER := QEVENTLINK[NNEW];
    if CURRENT = EVENTHEAD
      then
    begin
      QEVENTLINK[NNEW] := EVENTHEAD;
      EVENTHEAD := NNEW
    end
    else
    begin
      QEVENTLINK[NNEW] := QEVENTLINK[OLD];
      QEVENTLINK[OLD] := NNEW
    end;
    QEVENT[NNEW] := DEV;
    QEVENTPRTY[NNEW] := PRTY;
    QEVENTTIME[NNEW] := TIME
  end
  else
  begin
    WRITELN(BELL, '? EVENT-LIST OVER-FLOWED ');
    SETMODE(false, ZRUNMODE)
  end
end ENTEREVENT;
procedure export ENTERINTERRUPT(INTVEC,PRTY:integer);
  % ENTERS THE MAXIMUM OF IQSIZE OF INTERRUPTS IN THE QINT
  ERRUPT LIS
   T \
  var
    CURRENT, NNEW: integer;
```

```
  begin
    CURRENT := INTHEAD; NNEW := 0; OLD := 0;
    while CURRENT >= 0
      do
      if PRTY < QINTPRTY[CURRENT]
        then
      begin
        OLD := CURRENT;
        CURRENT := QINTLINK[CURRENT]
      end
      else exit;
    if FREEINTPOINTER >= 0
      then
    begin
      NNEW := FREEINTPOINTER;
      FREEINTPOINTER := QINTLINK[FREEINTPOINTER];
      if CURRENT = INTHEAD
        then
      begin
        QINTLINK[NNEW] := INTHEAD;
        INTHEAD := NNEW
      end
      else
      begin
        QINTLINK[NNEW] := QINTLINK[OLD];
        QINTLINK[OLD] := NNEW
      end;
      QINTVEC[NNEW] := INTVEC;
      QINTPRTY[NNEW] := PRTY
    end
    else
    begin
      WRITELN(BELL, ' ? INTERRUPT-LIST OVER-FLOWED ');
      SETMODE(false, ZRUNMODE)
    end
  end ENTERINTERRUPT;
Procedure export IODONE(event :integer);
  var
    CH: char;
  begin
    case event  of
      HSR:
        begin
          if not (BIT(HSRSTATUS, 21))
            then
          begin
            READ(CH);  % needs clarification
            HSRBUF := ORD(CH) * ORD('0');
            if not (HSRMODE) and (HSRBUF = 032B)
                then
```

```
      begin
        SETBIT(21, true, HSRSTATUS);
        % close input file \
        WRITELN(BELL, ' END - OF - LINE ON HSR ');
        SETMODE(false, ZRUNMODE)
      end
      else
      begin
        SETBIT(25, false, HSRSTATUS);
        SETBIT(29, true, HSRSTATUS);
        if BIT(HSRSTATUS, 30)
          then ENTERINTERRUPT(HSRLOCATION, HSRPRTY)
      end
    end
    else
      if BIT(HSRSTATUS, 30)
        then ENTERINTERRUPT(HSRLOCATION, HSRPRTY)
  end;
HSP:
  begin
    if not (BIT(HSRSTATUS, 21))
      then
      if not (HSPMODE) and (HSPBUF = 032B)
        then
      begin
        SETBIT(21, true, HSPSTATUS);
        % close output file -needs clarification
        \
      end
      else
      begin
        WRITE(HSPBUF);
        SETBIT(29, true, HSPSTATUS);
        if BIT(30, HSPSTATUS)
          then
        begin
          ENTERINTERRUPT(HSPLOCATION, HSPPRTY);
          SETBIT(30, false, HSPSTATUS)
        end
      end
    else
    begin
      if BIT(HSPSTATUS, 30)
        then
      begin
        ENTERINTERRUPT(HSPLOCATION, HSRPRTY);
        SETBIT(30, false, HSPSTATUS)
      end
    end
  end;
```

```
      KBD:
        begin
          KBDBUF := ORD(KBDCHR);
          SETBIT(25, false, KBDSTATUS);
          SETBIT(29, true, KBDSTATUS);
          if BIT(KBDSTATUS, 30)
            then ENTERINTERRUPT(KBDLOCATION, KBDPRTY)
        end;
      TTY:
        begin
          WRITE(TTYBUF);
          SETBIT(29, true, TTYSTATUS);
          if BIT(TTYSTATUS, 30)
            then
            begin
              ENTERINTERRUPT(TTYLOCATION, TTYPRTY);
              SETBIT(30, false, TTYSTATUS)
            end
        end;
      CLK:
        begin
          SETBIT(29, true, CLKSTATUS);
          CLKFLAG := true;
          if BIT(CLKSTATUS, 30)
            then
            begin
              ENTERINTERRUPT(CLKLOCATION, CLKPRTY);
              SETBIT(29, false, CLKSTATUS)
            end
        end;
      otherwise: WRITELN(BELL,
        ' *** ILLEGAL DEVICE IN IODONE ***')
    end
  end IODONE;
procedure export SEREVENT;
  var
    I: integer;
  begin
    I := 0;
    while (EVENTHEAD >= 0) and (REALTIME >= QEVENTTIME[
        EVENTHEAD])
          do
    begin
      I := EVENTHEAD;
      EVENTHEAD := QEVENTLINK[EVENTHEAD];
      QEVENTLINK[I] := FREEEVENTPOINTER;
      FREEEVENTPOINTER := I;
      IODONE(QEVENT[I])
    end
  end SEREVENT;
```

```
procedure export SERTRAP(LOCATION:integer);
  var
    REQWORD: integer;
    R1: WORDTYPE(DECWORD);
    R2: WORDTYPE(IDCWORD);
  begin
    for I in TRAP to BUSTRAP
      do SETTRAP(false, I);
    GETWORD(REG[1] + 2, RIGHT, WANTEDWORD);
    WRITEWORD(REG1, WANTEDWORD);
    WRITEWORD(REG[1], PSVAL);
    GETWORD(REG[1] + 2, RIGHT, WANTEDWORD);
    WRITEWORD(REG1, WANTEDWORD);
    WRITEWORD(REG[1], REG[0]);
    MREADWORD(LOCATION, REQWORD);
    WRITEWORD(REG0, REQWORD);
    MREADWORD(LOCATION + 2, REQWORD);
    WRITEWORD(PS, REQWORD);
    SETTIME(REALTIME + TRAPTIME)
  end SERTRAP;
procedure export SERINTERRUPT;
  var
    I, CPUPRTY: integer;
    R1: WORDTYPE(DECWORD);
    R9: WORDTYPE(PRIORITY);
  begin
    I := 0;
    R1.FULLWORD := PSVAL;
    R9 := WORDTYPE(PRIORITY); R1;
    CPUPRTY := R9.PRTYMASK;
    %WRITELN('PSVAL=', PSVAL, 'CPUPRTY=', CPUPRTY, ' INTH\
    EAD=', INT
     HEAD);\
    %WRITELN('QINTPRTY[INTHEAD]=', QINTPRTY[INTHEAD]);\
    if INTHEAD >= 0
      then
    begin
      if wait and (CPUPRTY >= QINTPRTY[INTHEAD])
        then
      begin
        CPUPRTY := QINTPRTY[INTHEAD] - 1;
        SETMASKPRTY(PS, CPUPRTY)
      end;
      if CPUPRTY < QINTPRTY[INTHEAD]
        then
      begin
        SETFLAG(false, ZWAIT);
        I := INTHEAD;
        INTHEAD := QINTLINK[INTHEAD];
        QINTLINK[I] := FREEINTPOINTER;
```

```
        FREEINTPOINTER := I;
        SETMODE(true, ZSPMODE);
        SERTRAP(QINTVEC[I])
      end
    end
    else
      if wait and (EVENTHEAD >= 0)
        then SETTIME(QEVENTTIME[EVENTHEAD])
  end SERINTERRUPT;
procedure export DOTRAP;
  var
    I: TRAPTYPE;
  begin
    while TRAPFLAG do
      if BUSTRAPFLAG
        then SERTRAP(ILLBUS)
      else
        if RTRAPFLAG
          then SERTRAP(RESLOCATION)
        else
          if ILLTRAPFLAG
            then SERTRAP(ILLBUS)
          else
            if BTRAPFLAG
              then SERTRAP(BPTLOCATION)
            else
              if STRAPFLAG
                then SERTRAP(STKLOCATION)
              else SETTRAP(false, TRAP)
  end DOTRAP;
procedure export IOREAD(ADDRESS:integer;
  var TEMP:integer);
  begin
    case ADDRESS of
      HSR: TEMP := HSRSTATUS;
      HSR1: GETBTE(HSRSTATUS, BYT1, TEMP);
      HSR2:
        begin
          SETBIT(29, false, HSRSTATUS);
          TEMP := HSRBUF
        end;
      HSR3:
        begin
          GETBTE(HSRBUF, BYT1, TEMP);
          SETBIT(7, false, HSRSTATUS)
        end;
      HSP: TEMP := HSPSTATUS;
      HSP1: GETBTE(HSPSTATUS, BYT1, TEMP);
      KBD: TEMP := KBDSTATUS;
      KBD1: GETBTE(KBDSTATUS, BYT1, TEMP);
```

```
      KBD2:
        begin
          SETBIT(29, false, KBDSTATUS);
          TEMP := KBDBUF
        end;
      KBD3:
        begin
          SETBIT(29, false, KBDSTATUS);
          GETBTE(KBDBUF, BYT1, TEMP)
        end;
      TTE: TEMP := TTYSTATUS;
      TTE1: GETBTE(TTYSTATUS, BYT1, TEMP);
      CLK: TEMP := CLKSTATUS;
      CLK1: GETBTE(CLKSTATUS, BYT1, TEMP);
      otherwise:
        begin
          SETTRAP(true, TRAP);
          SETTRAP(true, BUSTRAP);
          if DEBUGGING
            then
          begin
            WRITELN(BELL,
              'TIME OUT FOR DEVICE ADDRESS WHILE READING',
                    :10,
                    ADDRESS);
            TEMP := 0
          end
        end
    end
  end IOREAD;
procedure export IOWRITE(ADDRESS,DATA:integer);
  var
    I: boolean;
    R1: WORDTYPE(DECWORD);
    R3: WORDTYPE(BYTE);
    R8: WORDTYPE(BITS);
  begin
    I := false;
    case ADDRESS of
      HSR:
        begin
          I := BIT(DATA, 30);
          if (not (BIT(HSRSTATUS, 30))) and (I) and ((BIT
            HSRSTATUS,
            29)) or (BIT(HSRSTATUS, 21)))
            then ENTERINTERRUPT(HSRLOCATION, HSRPRTY);
          SETBIT(30, I, HSRSTATUS);
          if BIT(DATA, 36)
            then
```

```
      begin
        HSRBUF := 0;
        SETBIT(29, false, HSRSTATUS);
        SETBIT(25, true, HSRSTATUS);
        ENTEREVENT(HSR, REALTIME + HSRTIME, HSRPRTY)
      end;
    SETBIT(21, BIT(DATA, 21), HSRSTATUS)
  end;
HSR1: ;
HSR:
  begin
    I := BIT(DATA, 30);
    if (not (BIT(HSRSTATUS, 30))) and (I) and ((BIT(
       HSPSTATUS,
       29)) or (BIT(HSPSTATUS, 21)))
      then ENTERINTERRUPT(HSPLOCATION, HSRPRTY)
    else SETBIT(30, I, HSRSTATUS);
    SETBIT(21, BIT(DATA, 21), HSPSTATUS)
  end;
HSP1: ;
HSP2:
  begin
    HSPBUF := DATA;
    SETBIT(29, I, HSPSTATUS);
    ENTEREVENT(HSP, REALTIME + HSPTIME, HSPPRTY)
  end;
HSP3: ;
KBD:
  begin
    I := BIT(DATA, 30);
    if (not (BIT(KBDSTATUS, 30))) and (I) and (BIT(
       KBDSTATUS,
       29))
      then ENTERINTERRUPT(KBDLOCATION, KBDPRTY);
    SETBIT(30, I, KBDSTATUS)
  end;
KBD1: ;

TTY:
  begin
    I := BIT(DATA, 30);
    if (not (BIT(TTYSTATUS, 30))) and (I) and (BIT(
       TTYSTATUS,
       29))
      then ENTERINTERRUPT(TTYLOCATION, TTYPRTY)
    else SETBIT(30, I, TTYSTATUS)
  end;
TTY1: ;
TTY2:
  begin
```

```
        TTYBUF := DATA;
        SETBIT(29, false, TTYSTATUS);
        ENTEREVENT(TTP, REALTIME + TTYTIME, TTYPRTY)
      end;
    TTP3: ;
    CLK:
      begin
        I := BIT(DATA, 30);
        if (not (BIT(CLKSTATUS, 30)))
            and (I) and (BIT(CLKSTATUS, 29))

          then
        begin
          ENTERINTERRUPT(CLKLOCATION, CLKPRTY);
          SETBIT(29, false, CLKSTATUS)
        end;
        SETBIT(30, I, CLKSTATUS)
      end;
    CLK1: ;
    otherwise:
      begin
        SETTRAP(true, BUSTRAP);
        SETTRAP(true, TRAP);
        if DEBUGGING
          then WRITELN(BELL,

'TIME OUT FOR DEVICE ADDRESS WHILE WRITING', 0: 10,
                          ADDRESS);
        end
    end;
  end IOWRITE;
Procedure export KBDSIM;
  var
    CH: char;
  begin
    CH := CHR(0);
    READ(CH);
    if CH <> CHR(0)
      then
    begin
      if CH = CHR(030B) % ctrl x \
      then SETMODE(false, ZRUNMODE)
      else
      begin
        KBDCHR := CH;
        SETBIT(25, true, KBDSTATUS);
        SETBIT(29, false, KBDSTATUS);
        ENTEREVENT(KBD, REALTIME + KBDTIME, KBDPRTY);
        if CH = CHR(015B) % CARR RETURN   & LF \
```

```
03600          ENTEREVENT(KBD, REALTIME + KBDTIME, KBDPRTY);
03700          if CH = CHR(015B) % CARR RETURN  & LF \
03800            then READ(CH)
03900        end
04000      end
04100    end KBDSIM;
04200  procedure export CLKSIM;
04300    begin
04400      if CLKFLAG
04500        then
04600      begin
04700        ENTEREVENT(CLK, REALTIME + LINEFREQUENCY, CLKPRTY);
04800        CLKFLAG := false
04900      end
05000    end CLKSIM;
05100  begin % reals \
05200    KBDCHR := ' ';
05300  end REALS.
```

```
**END** User KRISHNA [70,105]  Job  REALS seq. 2178 Date 18-May-82 12:43:47

**END** User KRISHNA [70,105]  Job  REALS seq. 2178 Date 18-May-82 12:43:47

                    * * * L P T S P L  R u n  L o g * * *

12:42:50 LPDAT  [LPTLSJ  LPTSPL version 102(2263) running on LPT101, 18-May-8
12:42:50 LPDAT  [LPTSJS  Starting Job REALS, Seq #2178, request created at 18
12:42:52 LPMSG  [LPTSTF  Starting File DSKC:REALS.DIP<057>[70,105,OVG]]
12:43:46 LPMSG  [LPTFPF  Finished Printing File DSKC:REALS.DIP<057>[70,105,OV
12:43:47 LPSUM  Spooler runtime 1 seconds, 11 KCS, 32 disk reads, 13 pages pr

**END** User KRISHNA [70,105]  Job  REALS seq. 2178 Date 18-May-82 12:43:47

**END** User KRISHNA [70,105]  Job  REALS seq. 2178 Date 18-May-82 12:43:47

**END** User KRISHNA [70,105]  Job  REALS seq. 2178 Date 18-May-82 12:43:47

**END** User KRISHNA [70,105]  Job  REALS seq. 2178 Date 18-May-82 12:43:47

**END** User KRISHNA [70,105]  Job  REALS seq. 2178 Date 18-May-82 12:43:47

**END** User KRISHNA [70,105]  Job  REALS seq. 2178 Date 18-May-82 12:43:47

**END** User KRISHNA [70,105]  Job  REALS seq. 2178 Date 18-May-82 12:43:47

**END** User KRISHNA [70,105]  Job  REALS seq. 2178 Date 18-May-82 12:43:47

**END** User KRISHNA [70,105]  Job  REALS seq. 2178 Date 18-May-82 12:43:47

**END** User KRISHNA [70,105]  Job  REALS seq. 2178 Date 18-May-82 12:43:47

**END** User KRISHNA [70,105]  Job  REALS seq. 2178 Date 18-May-82 12:43:47
```

START User KRISHNA [70,103] Job SIMRUN Seq. 1641 Date 10-May-82 17:47:59

START User KRISHNA [70,103] Job SIMRUN Seq. 1641 Date 10-May-82 17:47:59

```
K   K  RRRR     III     SSSS  H   H  N   N   AAA
K  K   R   R     I     S      H   H  NN  N  A   A
K K    R   R     I     S      H   H  N N N  A   A
KK     RRRR      I      SSS   HHHHH  N N N  A   A
K K    R R       I         S  H   H  N  NN  AAAAA
K  K   R  R      I         S  H   H  N   N  A   A
K   K  R   R   III    SSSS   H   H  N   N  A   A
```

START User KRISHNA [70,103] Job SIMRUN Seq. 1641 Date 10-May-82 17:47:59

START User KRISHNA [70,103] Job SIMRUN Seq. 1641 Date 10-May-82 17:47:59

START User KRISHNA [70,103] Job SIMRUN Seq. 1641 Date 10-May-82 17:47:59

START User KRISHNA [70,103] Job SIMRUN Seq. 1641 Date 10-May-82 17:47:59

START User KRISHNA [70,103] Job SIMRUN Seq. 1641 Date 10-May-82 17:47:59

START User KRISHNA [70,103] Job SIMRUN Seq. 1641 Date 10-May-82 17:47:59

START User KRISHNA [70,103] Job SIMRUN Seq. 1641 Date 10-May-82 17:47:59

```
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000001111111111222222222233333333334444444445555555556666666667777777
1234567890123456789012345678901234567890123456789012345678901234567890123456
```

```
module SIMRUN options(VERSION = ASIM, DEFAULTCXT)
  include REALS, DEKODE, IOSYS;
  const export
    MAXINS = 76;
    MAXBREAK = 7;
  var export
    ADRFLAG, INSFLAG, SINGLE: boolean;
    ADRBREAK, INSBREAK: array [0 .. MAXBREAK] of integer;
    INSCOUNT: array [0 .. MAXINS] of integer;
    MEMNIC: array [0 .. MAXINS] of string(5);
  var
    OLDPC, INSREG, OPCODE: integer;
  type export
    BREAKTYPE = (ZADRBREAK, ZINSBREAK);
    BRKFLAG = (ZINSFLAG, ZADRFLAG, ZSINGLE);
  const
    FETCHTIME = 1;
  procedure export INITINSCOUNT;
    begin
      for I in 0 to MAXINS
        do INSCOUNT[I] := 0
    end {INITINSCOUNT};
  procedure export SETBREAK(BREAK:BREAKTYPE;
                            BREAKNUM,NEWVALUE:integer);
    begin
      case BREAK of
        ZADRBREAK: ADRBREAK[BREAKNUM] := NEWVALUE;
        ZINSBREAK: INSBREAK[BREAKNUM] := NEWVALUE
      end
    end SETBREAK;
  procedure export SETBRKFLAG(FLAG:BRKFLAG;
    NEWVALUE:boolean);
    begin
      case FLAG of
        ZINSFLAG: INSFLAG := NEWVALUE;
        ZADRFLAG: ADRFLAG := NEWVALUE;
        ZSINGLE: SINGLE := NEWVALUE
      end
    end SETBRKFLAG;
  procedure export SIMULATE;
    procedure RUN;
      var
        ADR, WANTEDWORD, SRCADD, REQUIRED: integer;
        INSTRUCTION: TYPEOPCODE;
      begin %******* RUN *******\
        %WRITELN('  Entered SIMRUN');\
        SETADR(0, ZSRC);
        SETADR(0, ZDSTADD);
        SETFLAG(false, ZBYTEFLAG);
        %******* CHECK FOR BREAK - POINT TRAP *******\
        SETTRAP(BIT(PSVAL, 32), BTRAP);
        %******* FETCH STATE *******\
        %WRITELN('Reg[0]=', 0: 10, REG[0]);\
        READWORD(REG[0], INSREG);
        %WRITELN('  Insreg= ', 0: 10, INSREG);\
        GETWORD(REG[0] + 2, RIGHT, WANTEDWORD);
        RECSET(WANTEDWORD, 0);
        %WRITELN('Reg[0]= ', 0: 10, REG[0]);\
        SETTIME(REALTIME + FETCHTIME);
```

```
      if TRAPFLAG
        then return ;
      $******* DECODE STATE ******\
      OPCODE := DECODE(INSREG);
      WRITELN('Insreg= ', 0: 10, INSREG, ' Opcode= ',
              OPCODE);
      if TRAPFLAG

        then return ;
      if (0 < OPCODE) and (OPCODE <= MAXINS)
        then INSCOUNT[OPCODE - 1] := INSCOUNT[OPCODE - 1]
             + 1
      else
      begin
        WRITE(BELL,

'OPCODE EXCEEDED 75 IN RUN PROCEDURE HENCE ABORTED');
        return
      end;
      $******* SOURCE OPERAND EVALUATION ******\
      if OPCODE <= 14 then
      begin
        OPERANDS(INSREG, SOURCE, ADD);
        SRCADD := ADD;
        if ADRFLAG
          then
          for K in 0 to MAXBREAK
              do
            if ADRBREAK[K] = ADD
              then
            begin
              WRITELN(BELL, 'Address Break', 0: 2, K,
              ' for', 0: 6, ADD, ' at PC ', 0: 6, OLDPC);
              SETMODE(false, ZRUNMODE)
            end;
        REQUIRED := 0;
        $WRITELN('Srcadd=', 0: 10, SRCADD);\
        if BYTEFLAG
          then READBYTE(SRCADD, REQUIRED)
        else READWORD(SRCADD, REQUIRED);
        SETADR(REQUIRED, ZSRC);
        $WRITELN('Src=', 0: 10, SRC);\
        SETTIME(REALTIME + SRCTIME);
        $WRITELN('Srctime=', SRCTIME, ' Realtime=', REALTI
        ME);\
        if TRAPFLAG
          then return
      end;
      $******* destination operand evaluation *******\
      if OPCODE <= 17 then
      begin
        OPERANDS(INSREG, DESTINATION, ADD);
        SETADR(ADD, ZDSTADD);
        if ADRFLAG
          then
          for K in 0 to MAXBREAK
              do
            if ADRBREAK[K] = ADD
```

```
                then
              begin
                WRITELN(BELL, 'Address Break', 0: 2, K,
                'for', 0: 6, ADD, 'at pc ', 0: 6, OLDPC);
                SETMODE(false, ZRUNMODE)
              end;
            SETTIME(REALTIME + DSTIME);
            {WRITELN('Dstadd=', 0: 16, DSTADD, 'Realtime=', RE
            ALTIME),\
            if TRAPFLAG
              then return
          end;
          {********  EXECUTE STATE    *******\
          INSTRUCTION := TYPE@OPCODE: OPCODE;
          EXECUTE(INSTRUCTION);
          SETTIME(REALTIME + INSTIME);

          if TRAPFLAG
            then return ;
          {********  RECORD IF ANY BREAK-POINT REQUEST  ********
          \
          SETTRAP(DTRAPFLAG, DTRAP);
          return
        end RUN;
      begin {********  SIMULATE*******\
        loop
          CLKSIM;
          KBDSIM;
          SEREVENT;
          {WRITELN('Wait=', wait );\
          if not wait
            then
            begin
              OLDPC := REG[0];
              RUN
            end;
          if TRAPFLAG
            then DOTRAP;
          SERINTERRUPT;
          if SINGLE
            then
            begin
              WRITELN;
              WRITELN(

'Nemnic
PCval    In
str  Source  Dstadd MQval  SPval   BSval   SCval ');
              WRITELN(NEMNIC[OPCODE], '    ', 0: 5, OLDPC,
                      '    ', 0: 6, INSREG, '    ', 0: 6, SRC,
```

```
        ' ', 0: 6, DSTADD,
                ', 0: 6, SIGVAL, '    ', 0: 6, REG[1],
        ', W: 5, PSVAL, '    ', W: 6, SCVAL);
        SETFLAG(false, ZWAIT);
        return
      end;
      if HALTFLAG
        then
      begin
        WRITELN(BELL, ' Halt at PC ', 0: 6, OLDPC);
        return
      end;
      if not RUNMODE
        then return ;
      if INSFLAG
        then
        for I in 0 to MAXBREAK
          do
          if (REG[0] = INSBREAK[I])
            then
          begin
            WRITELN(BELL, 'Instruction break ', 0: 2, I,
            'at PC ', 0: 6, REG[0], 'oldPC', 0: 6, OLDPC);
            return
          end
    end
  end SIMULATE;
begin ***** simrun *****\
  MEMNIC[0] := 'ERR  '; MEMNIC[1] := 'IOR  ';
  MEMNIC[2] := 'IORB '; MEMNIC[3] := 'CMP  ';
  MEMNIC[4] := 'CMPB '; MEMNIC[5] := 'AND  ';
  MEMNIC[6] := 'ANDB '; MEMNIC[7] := 'MUL  ';
  MEMNIC[8] := 'DIV  '; MEMNIC[9] := 'MOV  ';
  MEMNIC[10] := 'MOVB '; MEMNIC[11] := 'ADD  ';
  MEMNIC[12] := 'SUB  '; MEMNIC[13] := 'CLB  ';
  MEMNIC[14] := 'CLBB '; MEMNIC[15] := 'CAL  ';
  MEMNIC[16] := 'CLR  '; MEMNIC[17] := 'CLRB ';
  MEMNIC[18] := 'TST  '; MEMNIC[19] := 'TSTB ';
  MEMNIC[20] := 'INC  '; MEMNIC[12] := 'INCB ';
  MEMNIC[22] := 'CAD  '; MEMNIC[23] := 'CADB ';
  MEMNIC[24] := 'DEC  '; MEMNIC[25] := 'DECB ';
  MEMNIC[26] := 'CSB  '; MEMNIC[27] := 'CSBB ';
  MEMNIC[28] := 'COM  '; MEMNIC[29] := 'COMB ';
  MEMNIC[30] := 'NOT  '; MEMNIC[31] := 'NOTB ';
  MEMNIC[32] := 'SHR  '; MEMNIC[33] := 'SHRB ';
  MEMNIC[34] := 'ESR  '; MEMNIC[35] := 'ELR  ';
  MEMNIC[36] := 'SHL  '; MEMNIC[37] := 'SHLB ';
  MEMNIC[38] := 'ELL  '; MEMNIC[39] := 'ENM  ';
  MEMNIC[40] := 'CIR  '; MEMNIC[41] := 'CIRB ';
  MEMNIC[42] := 'CIL  '; MEMNIC[43] := 'CILB ';
```

```
18800       NEMNIC[44] := 'ECL  '; NEMNIC[45] := 'ECR  ';
18850       NEMNIC[46] := 'JMP  '; NEMNIC[47] := 'EXB  ';
18900       NEMNIC[48] := 'DTN  '; NEMNIC[49] := 'TNE  ';
18950       NEMNIC[50] := 'TGE  '; NEMNIC[51] := 'TGT  ';
19000       NEMNIC[52] := 'TRA  '; NEMNIC[53] := 'TEQ  ';
19050       NEMNIC[54] := 'TLT  '; NEMNIC[55] := 'TLE  ';
19100       NEMNIC[56] := 'TPL  '; NEMNIC[57] := 'TOC  ';
19150       NEMNIC[58] := 'TCC  '; NEMNIC[59] := 'THI  ';
19200       NEMNIC[60] := 'TMI  '; NEMNIC[61] := 'TOS  ';
19250       NEMNIC[62] := 'TCS  '; NEMNIC[63] := 'TLS  ';
19300       NEMNIC[64] := 'SVC  '; NEMNIC[65] := 'XPR  ';
19350       NEMNIC[66] := 'TRP  '; NEMNIC[67] := 'SETC ';
19400       NEMNIC[68] := 'CLRC '; NEMNIC[69] := 'RES  ';
19450       NEMNIC[70] := 'HLT  '; NEMNIC[71] := 'WFI  ';
19500       NEMNIC[72] := 'REI  '; NEMNIC[73] := 'BPT  ';
19550       NEMNIC[74] := 'MCL  '; NEMNIC[75] := 'ERR  ';
19600       OLDPC := 0;
19700       for I in 0 to MAXINS
19800         do INSCOUNT[I] := 0
19900     end SIMRUN.
```

```
**END** User KRISHNA [70,103]  Job  SIMRUN Seq. 1641 Date 10-May-82 17:48:18

**END** User KRISHNA [70,103]  Job  SIMRUN Seq. 1641 Date 10-May-82 17:48:18

                    * * * L P T S P L  R u n  L o g * * *

17:47:58 LPDAT  [LPTLSJ  LPTSPL version 102(2263) running on LPT101, 10-May-8
17:47:58 LPDAT  [LPTSJS  Starting Job SIMRUN, Seq #1641, request created at 1
17:47:59 LPMSG  [LPTSTF  Starting File DSKC:QVG101.LPT<077>[3,3](SIMRUN)]
17:48:18 LPMSG  [LPTFPF  Finished Printing File DSKC:QVG101.LPT<077>[3,3](SIM
17:48:18 LPSUM  Spooler runtime 8 Seconds, 6 KCS, 18 disk reads, 5 pages prin

**END** User KRISHNA [70,103]  Job  SIMRUN Seq. 1641 Date 10-May-82 17:48:18

**END** User KRISHNA [70,103]  Job  SIMRUN Seq. 1641 Date 10-May-82 17:48:18

**END** User KRISHNA [70,103]  Job  SIMRUN Seq. 1641 Date 10-May-82 17:48:18

**END** User KRISHNA [70,103]  Job  SIMRUN Seq. 1641 Date 10-May-82 17:48:18

**END** User KRISHNA [70,103]  Job  SIMRUN Seq. 1641 Date 10-May-82 17:48:18

**END** User KRISHNA [70,103]  Job  SIMRUN Seq. 1641 Date 10-May-82 17:48:18

**END** User KRISHNA [70,103]  Job  SIMRUN Seq. 1641 Date 10-May-82 17:48:18

**END** User KRISHNA [70,103]  Job  SIMRUN Seq. 1641 Date 10-May-82 17:48:18

**END** User KRISHNA [70,103]  Job  SIMRUN Seq. 1641 Date 10-May-82 17:48:18

**END** User KRISHNA [70,103]  Job  SIMRUN Seq. 1641 Date 10-May-82 17:48:18

**END** User KRISHNA [70,103]  Job  SIMRUN Seq. 1641 Date 10-May-82 17:48:18
```

```
12345678901234567890123456789012345678901234567890123456789012345678901234567

*START* User KRISHNA [70,105] Job DEKODE Seq. 2169 Date 18-May-82 12:41:28

*START* User KRISHNA [70,105] Job DEKODE Seq. 2169 Date 18-May-82 12:41:28


K   K  RRRR    III    SSSS  H   H  N   N   AAA
K  K   R   R    I    S      H   H  NN  N  A   A
K K    R   R    I    S      H   H  N N N  A   A
KKK    RRRR     I     SSS   HHHHH  N  NN  A   A
K K    R R      I        S  H   H  N   N  AAAAA
K  K   R  R     I        S  H   H  N   N  A   A
K   K  R   R   III   SSSS   H   H  N   N  A   A


*START* User KRISHNA [70,105] Job DEKODE Seq. 2169 Date 18-May-82 12:41:28

*START* User KRISHNA [70,105] Job DEKODE Seq. 2169 Date 18-May-82 12:41:28

*START* User KRISHNA [70,105] Job DEKODE Seq. 2169 Date 18-May-82 12:41:28

*START* User KRISHNA [70,105] Job DEKODE Seq. 2169 Date 18-May-82 12:41:28

*START* User KRISHNA [70,105] Job DEKODE Seq. 2169 Date 18-May-82 12:41:28

*START* User KRISHNA [70,105] Job DEKODE Seq. 2169 Date 18-May-82 12:41:28

*START* User KRISHNA [70,105] Job DEKODE Seq. 2169 Date 18-May-82 12:41:28

*START* User KRISHNA [70,105] Job DEKODE Seq. 2169 Date 18-May-82 12:41:28

*START* User KRISHNA [70,105] Job DEKODE Seq. 2169 Date 18-May-82 12:41:28

00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000011111111112222222222333333333344444444445555555555666666666677777777
12345678901234567890123456789012345678901234567890123456789012345678901234567
```

```
module DEKODE options(VERSION := NSIM, DEFAULTCXT)
  include EXEC, IOSYS;
  type export
    AREATYPE = (SOURCE, DESTINATION);
  var export
    DSTTIME, SRCTIME: integer;
  const
    REGADR = 177740B;
  type
    VIEWTYPE = (DECWORD, TDCWORD, TDCBYTES, BYTE, BITS,
                FOURMASK,
                BYTESELECT, OCTMASK, COZ);
    WORDTYPE =
      packed record
        case VIEWTYPE of
          DECWORD: (FULLWORD: integer);
          TDCWORD: (DUMMY1TDC: 0 .. 17B;
                    LEFTWORD, RIGHTWORD: 0 .. 177777B);
          TDCBYTES: (JUNK1: 0 .. 3B;
          TDCBYTE1, TDCBYTE0: 0 .. 377B;
                    JUNK2: 0 .. 3B;
          TDCBYTE3, TDCBYTE2: 0 .. 377B);
          BYTE: (DUMMYBYTE: 0 .. 17B;
                 BYTE3, BYTE2, BYTE1, BYTE0: 0 .. 377B);
          BITS: (BITVAL: packed array [1 .. 36] of boolean)
      end WORDTYPE;
  var
    REGTIME: array [0 .. 7] of integer;
    SREG, DREG, SMODE, DMODE: integer;
  procedure export INITTIME;
    begin
      DSTTIME := 0; SRCTIME := 0
    end INITTIME;
  function ADDRESS(MODE,NREG:integer) return integer;
    var
      STEP, M, SHIFTED, K, REGWORD, TEMP, WANTEDWORD:
      integer;
      I, BOOL, IRETURN: boolean;
      R1: WORDTYPE(DECWORD);
      R2: WORDTYPE(TDCWORD);
      R8: WORDTYPE(BITS);
    begin
      I := BIT(MODE, 34);
      M := LOGAND(MODE, 3B);
      %WRITELN(' I=', I, ' M=', M);\
      BOOL := (I) or (not BYTEFLAG) or (NREG < 2);
      if BOOL
        then STEP := 2
      else STEP := 1;
      SETMODE(false, ZSPMODE);
      SETMODE(false, ZREGMODE);
      case M of
        0:
          begin
            SETMODE((MODE = 0), ZREGMODE);
            TEMP := NREG * 2 + REGADR
          end;
        1:
          begin
```

```
        TEMP := REG[NREG];
        GETWORD(REG[NREG] + STEP, RIGHT, WANTEDWORD);
        REGSET(WANTEDWORD, NREG);
        % WRITELN('reg[nreg]=', 0: 10, REG[NREG])\
      end;
    2:
      begin


        GETWORD(REG[NREG] + STEP, RIGHT, WANTEDWORD);
        REGSET(WANTEDWORD, NREG);
        SETMODE((NREG = 1), ZSPMODE);
        TEMP := REG[NREG]
      end;
    3:
      begin
        READWORD(REG[0], REQWORD);
        GETWORD(REG[0] + 2, RIGHT, WANTEDWORD);
        REGSET(WANTEDWORD, 0);
        GETWORD(REQWORD + REG[NREG], RIGHT, TEMP)
      end
  end;
  %WRITELN('Temp=', 0: 10, TEMP);\
  if I
    then READWORD(TEMP, REQWORD)
  else REQWORD := TEMP;
  return REQWORD
end ADDRESS;
function export DECODE(INS:integer) return integer;
  var
    FIRST, SECOND, OCT3, OCT4, SHIFTED, OCT2, TEMP:
    integer;
    R1: WORDTYPE(DECWORD);
    R3: WORDTYPE(BYTE);
  begin % DECODE\
    RSHIFT(INS, 12, SHIFTED);
    FIRST := LOGAND(SHIFTED, 17B);
    RSHIFT(INS, 8, SHIFTED);
    SECOND := LOGAND(SHIFTED, 17B);
    if FIRST > 1
      then
    begin
      SETFLAG(BIT(FIRST, 36) and (FIRST <> 9) and (FIRST
              <> 13),
            ZBYTEFLAG);
      return FIRST = 1
    end
    else
      if FIRST = 1
        then
      begin
        GETBTE(INS, BYTO, TEMP);
        SETADR(TEMP, ZOFFSET);
        return SECOND + 48
      end
      else
      begin
        RSHIFT(INS, 6, SHIFTED);
        OCT3 := LOGAND(SHIFTED, 7B);
```

```
RSHIFT(INS, 9, SHIFTED);
OCT4 := LOGAND(SHIFTED, 7B);
case OCT4 of
  2:
    begin
      SETTRAP(true, RTRAP);
      SETTRAP(true, TRAP);
      if DEBUGGING
        then WRITELN(BELL, O: 6, INS,
                     ' ? RESERVED INSTRUCTION ');
      return 0
    end;
  3:
    begin
      SETADR(OCT3, ZSUBREG);
      return 15
    end;
  4:
    if (OCT3 = 2) or (OCT3 = 3)
      then
    begin
      SETTRAP(true, RTRAP);
      SETTRAP(true, TRAP);
      if DEBUGGING
        then WRITELN(BELL,
             ' ? RESERVED INSTRUCTION ', O: 6,
                  INS);
      return 0
    end
    else
    begin
      SETFLAG(BIT(OCT3, 36) and (OCT3 <> 7),
              ZBYTEFLAG);
      if OCT3 <= 1
        then return OCT3 + 40
      else return OCT3 + 38
    end;
  5:
    begin
      SETFLAG((OCT3 = 1) or (OCT3 = 5), ZBYTEFLAG)
      ;
      return OCT3 + 32
    end;
  6:
    begin
      SETFLAG(BIT(OCT3, 36), ZBYTEFLAG);
      return OCT3 + 16
    end;
  7:
    begin
```

```
    SETFLAG(BIT(OCT3, 36), ZBYTEFLAG);
    return OCT3 + 24
  end;
otherwise:
  if (SECOND <> 0)
    then return 63 + SECOND
  else
    if (OCT3 <> 0)
      then
      if (OCT3 = 1)
        then
      begin
        SETADR(LOGAND(INS, 37B), ZCOZSB);
        if BIT(INS, 31)
          then return 67
        else return 68
      end
      else return OCT3 + 44
    else
    begin
      RSHIFT(INS, 3, SHIFTED);
      OCT2 := LOGAND(SHIFTED, 7B);
      if OCT2 = 0
        then
        if INS < 5
          then return INS + 70
        else
        begin
          SETTRAP(true, RTRAP);
          SETTRAP(true, TRAP);
          if DEBUGGING
            then
          begin
            WRITELN(BELL,
            ' ? RESERVED INSTRUCTION ', 0: 6
                  , INS);
            return 0
          end
        end
      else
        if OCT2 = 1
          then
        begin
          SETADR(LOGAND(INS, 7B), ZSUBREG);
          return 69
        end
        else
        begin
          SETTRAP(true, RTRAP);
          SETTRAP(true, TRAP);
```

```
                    if DEBUGGING
                      then
                    begin
                      WRITELN(BELL,
                      ' ? RESERVED INSTRUCTION ', O:6
                            , INS);
                      return 0
                    end
                  end
              end
            end
          end {CASE\
      end DECODE;
Procedure export OPERANDS(INSREG:integer; AREA:AREATYPE;
                          var ADD:integer);
    var
      SMODE, DMODE, SHIFTED: integer;
      R1: WORDTYPE(DECWORD);
      R2: WORDTYPE(IDCWORD);
      R11: WORDTYPE(OCTMASK);
    begin
      if AREA = SOURCE then
      begin
        RSHIFT(INSREG, 9, SHIFTED);
        SMODE := LOGAND(SHIFTED, 7B);
        RSHIFT(INSREG, 6, SHIFTED);
        SREG := LOGAND(SHIFTED, 7B);
        SRCTIME := REGTIME[SMODE];
        {WRITELN('Smode= ', SMODE, ' Sreg= ', SREG);\
        ADD := ADDRESS(SMODE, SREG)
      end
      else
        if AREA = DESTINATION
              then
        begin
          RSHIFT(INSREG, 3, SHIFTED);
          DMODE := LOGAND(SHIFTED, 7B);
          DREG := LOGAND(INSREG, 7B);
          DSTTIME := REGTIME[DMODE];
          {WRITELN('Dmode= ', DMODE, ' Dreg= ', DREG);\
          ADD := ADDRESS(DMODE, DREG)
        end
    end OPERANDS;
begin {dekode \
    REGTIME[0] := 0; REGTIME[1] := 0016;
    REGTIME[2] := 0016;
    REGTIME[3] := 0032; REGTIME[4] := 0014;
    REGTIME[5] := 0030; REGTIME[6] := 0030;
    REGTIME[7] := 0046
end DEKODE.
```

```
**END** User KRISHNA [70,105]  Job  DEKODE Seq. 2169 Date 18-May-82 12:41:51
```



```
**END** User KRISHNA [70,105]  Job  DEKODE Seq. 2169 Date 18-May-82 12:41:51
```



```
* * * L P T S P L   R u n   L o g * * *

12:41:27 LPDAT  [LPTLSJ  LPTSPL version 102(2263) running on LPT101, 18-May-8
12:41:27 LPDAT  [LPTSJS  Starting Job DEKODE, Seq #2169, request created at 1
12:41:29 LPMSG  [LPTSTF  Starting File DSKC:DEKODE.DIP<057>[70,105,OVG]]
12:41:50 LPMSG  [LPTFPF  Finished Printing File DSKC:DEKODE.DIP<057>[70,105,O
12:41:51 LPSUM  Spooler runtime 0 Seconds, 6 KCS, 15 disk reads, 6 pages prin
```



```
**END** User KRISHNA [70,105]  Job  DEKODE Seq. 2169 Date 18-May-82 12:41:51
**END** User KRISHNA [70,105]  Job  DEKODE Seq. 2169 Date 18-May-82 12:41:51
**END** User KRISHNA [70,105]  Job  DEKODE Seq. 2169 Date 18-May-82 12:41:51
**END** User KRISHNA [70,105]  Job  DEKODE Seq. 2169 Date 18-May-82 12:41:51
**END** User KRISHNA [70,105]  Job  DEKODE Seq. 2169 Date 18-May-82 12:41:51
**END** User KRISHNA [70,105]  Job  DEKODE Seq. 2169 Date 18-May-82 12:41:51
**END** User KRISHNA [70,105]  Job  DEKODE Seq. 2169 Date 18-May-82 12:41:51
**END** User KRISHNA [70,105]  Job  DEKODE Seq. 2169 Date 18-May-82 12:41:51
**END** User KRISHNA [70,105]  Job  DEKODE Seq. 2169 Date 18-May-82 12:41:51
**END** User KRISHNA [70,105]  Job  DEKODE Seq. 2169 Date 18-May-82 12:41:51
**END** User KRISHNA [70,105]  Job  DEKODE Seq. 2169 Date 18-May-82 12:41:51
```

```
1234567890123456789012345678901234567890123456789012345678901234567
```

START User KRISHNA [70,105] Job MEMORY Seq. 2189 Date 18-May-82 12:43:56

START User KRISHNA [70,105] Job MEMORY Seq. 2189 Date 18-May-82 12:43:56

```
K   K  RRRR    III    SSSS  H   H  N   N   AAA
K  K   R   R    I    S      H   H  N   N  A   A
K K    R   R    I    S      H   H  NN  N  A   A
KKK    RRRR     I     SSS   HHHHH  N N N  A   A
K  K   R R      I        S  H   H  N  NN  AAAAA
K   K  R  R     I        S  H   H  N   N  A   A
K   K  R   R   III   SSSS   H   H  N   N  A   A
```

START User KRISHNA [70,105] Job MEMORY Seq. 2189 Date 18-May-82 12:43:56

START User KRISHNA [70,105] Job MEMORY Seq. 2189 Date 18-May-82 12:43:56

START User KRISHNA [70,105] Job MEMORY Seq. 2189 Date 18-May-82 12:43:56

START User KRISHNA [70,105] Job MEMORY Seq. 2189 Date 18-May-82 12:43:56

START User KRISHNA [70,105] Job MEMORY Seq. 2189 Date 18-May-82 12:43:56

START User KRISHNA [70,105] Job MEMORY Seq. 2189 Date 18-May-82 12:43:56

START User KRISHNA [70,105] Job MEMORY Seq. 2189 Date 18-May-82 12:43:56

START User KRISHNA [70,105] Job MEMORY Seq. 2189 Date 18-May-82 12:43:56

START User KRISHNA [70,105] Job MEMORY Seq. 2189 Date 18-May-82 12:43:56

```
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000011111111112222222222333333333344444444445555555555666666666677777777
12345678901234567890123456789012345678901234567890123456789012345678901234567
```

```
module MEMORY options(VERSION := NSIM, DEFAULTCXT)
  include UTILTY, GLOBAL, UNI, IOSYS;
  type
    VIEWTYPE = (DECWORD, BYTE, BYTESELECT);
    WORDTYPE =
      packed record
        case VIEWTYPE of
          DECWORD: (FULLWORD: integer);
          BYTE: (DUMMYBYTE: 0 .. 17B;
          BYTE3, BYTE2, BYTE1, BYTE0: 0 .. 377B);
          BYTESELECT: (IGNORE: 0 .. 177777777777B;
          BYTESELECTOR: 0 .. 3B)
      end WORDTYPE;
  var
    TDCMEM: array [0 .. MEMSIZE] of integer;
    AREA: SUBPROCEZR;
  procedure export INITTDCMEM;
    begin
      for I in 0 to MEMSIZE
        do TDCMEM[I] := 0
    end INITTDCMEM;
  procedure export MREADWORD(ADDRESS:integer;
                             var REQWORD:integer);
    var
      INDEX: integer;
    begin
      INDEX := ADDRESS div 2;
      if (INDEX > MEMSIZE)
        then
      begin
        WRITELN(BELL,

'Array index exceeded Memsize in MREADWORD procedure');
        return
      end
      else
      begin
        if BIT(ADDRESS, 35)
          then GETWORD(TDCMEM[INDEX], LEFT, REQWORD)
          else GETWORD(TDCMEM[INDEX], RIGHT, REQWORD)
      end
    end MREADWORD;
  procedure export MREADBYTE(ADDRESS:integer;
                             var REQBYTE:integer);
    var
      INDEX, VALUE: integer;
      R2: WORDTYPE(BYTE);
      R1: WORDTYPE(DECWORD);
      R4: WORDTYPE(BYTESELECT);
    begin
      INDEX := ADDRESS div 2;
      if (INDEX > MEMSIZE)
        then
      begin
        WRITELN(BELL,

' Array index exceeded Memsize in MREADBYTE Procedure');
        return
      end
```

```
      else
      begin
        R1.FULLWORD := ADDRESS;
        R4 := WORDTYPE(BYTESELECT); R1;
        case R4.BYTESELECTOR of
          0: GETBTE(TDCMEM[INDEX], BYT0, REQBYTE);
          1: GETBTE(TDCMEM[INDEX], BYT1, REQBYTE);


          2: GETBTE(TDCMEM[INDEX], BYT2, REQBYTE);
          3: GETBTE(TDCMEM[INDEX], BYT3, REQBYTE)
        end
      end
    end MREADBYTE;
  procedure export MRITEWORD(ADDRESS:integer; DATA:integer);
    var
      INDEX: integer;
    begin
      INDEX := ADDRESS div 2;
      if (INDEX > MEMSIZE) and (0 > INDEX)
        then
      begin
        WRITELN(BELL,

'Array index exceeded Memsize in MRITEWORD procedure');
        return
      end
      else
        if BIT(ADDRESS, 35)
          then SETWORD(LEFT, DATA, TDCMEM[INDEX])
        else SETWORD(RIGHT, DATA, TDCMEM[INDEX])
    end MRITEWORD;
  procedure export MRITEBYTE(ADDRESS:integer; DATA:integer);
    var
      INDEX, VALUE: integer;
      R1: WORDTYPE(DECWORD);
      R4: WORDTYPE(BYTESELECT);
    begin
      INDEX := ADDRESS div 2;
      if (INDEX > MEMSIZE)
        then
      begin
        WRITELN(BELL,

'Array index exceeded Memsize in MRITEBYTE procedure');
        return
      end
      else
      begin
        R1.FULLWORD := ADDRESS;
        R4 := WORDTYPE(BYTESELECT); R1;
        VALUE := R4.BYTESELECTOR;
        case VALUE of
          0: SETBYTE(BYT0, DATA, TDCMEM[INDEX]);
          1: SETBYTE(BYT1, DATA, TDCMEM[INDEX]);
          2: SETBYTE(BYT2, DATA, TDCMEM[INDEX]);
          3: SETBYTE(BYT3, DATA, TDCMEM[INDEX])
        end
      end
```

```
  end MRITEBYTE;
procedure WDCORREAD(ADDRESS:integer; var REGWORD:integer);
  var
    EXWORD: integer;
    FAULT: boolean;
  begin
    FAULT := false;
    if SPMODE then
    begin
      SETMODE(false, ZSPMODE);
      EXTEND(SLVAL, WORDS, EXWORD);
      if (ADDRESS <= EXWORD) or (ADDRESS <= 400B)
        then
      begin
        ERRORS(ZREADWORD, ZCORE, ADDRESS);
        FAULT := true
      end;
    end;
    if not FAULT
      then MREADWORD(ADDRESS, REGWORD)
  end WDCORREAD;
procedure BTCORREAD(ADDRESS:integer; var REGBYTE:integer);
  var
    EXWORD: integer;
    FAULT: boolean;
  begin
    FAULT := false;
    if SPMODE then
    begin
      SETMODE(false, ZSPMODE);
      EXTEND(SLVAL, WORDS, EXWORD);
      if (ADDRESS <= EXWORD) or (ADDRESS <= 400B)
        then
      begin
        ERRORS(ZREADBYTE, ZCORE, ADDRESS);
        FAULT := true
      end;
    end;
    if not FAULT
      then MREADBYTE(ADDRESS, REGBYTE)
  end BTCORREAD;
procedure WDCORWRITE(ADDRESS,DATA:integer);
  var
    FAULT: boolean;
    EXWORD: integer;
  begin
    FAULT := false;
    if SPMODE then
    begin
      SETMODE(false, ZSPMODE);
```

```
      EXTEND(SLVAL, WORDS, EXWORD);
      if (ADDRESS <= EXWORD) or (ADDRESS <= 400B)
        then
      begin
        ERRORS(ZWRITEWORD, ZCORE, ADDRESS);
        FAULT := true
      end;
    end;
    if not FAULT
      then MRITEWORD(ADDRESS, DATA)
  end WDCORWRITE;
Procedure BTCORWRITE(ADDRESS,DATA:integer);
  var
    FAULT: boolean;
    EXWORD: integer;
  begin
    FAULT := false;
    if SPMODE then
    begin
      SETMODE(false, ZSPMODE);
      EXTEND(SLVAL, WORDS, EXWORD);
      if (ADDRESS <= EXWORD) or (ADDRESS <= 400B) then
      begin
        SETTRAP(true, STRAP);
        SETTRAP(true, TRAP);
        FAULT := true;
        if DEBUGGING
          then ERRORS(ZWRITEBYTE, ZCORE, ADDRESS)
      end;
    end;
    if not FAULT
      then MRITEBYTE(ADDRESS, DATA)
  end BTCORWRITE;
procedure export READWORD(ADDRESS:integer;
                          var REGWORD:integer);
  begin
    REGWORD := 0;
    ADRUNIBUS(ADDRESS, AREA);
    case AREA of
      ZBUSERR: ERRORS(ZREADWORD, ZBUSERR, ADDRESS);
      ZODDADR: ERRORS(ZREADWORD, ZODDADR, ADDRESS);
      ZCPU: WDCPUREAD(ADDRESS, REGWORD);
      ZCORE: WDCORREAD(ADDRESS, REGWORD);
      ZINOUT: WDIOREAD(ADDRESS, REGWORD)
    end
  end READWORD;
procedure export READBYTE(ADDRESS:integer;
                          var REGBYTE:integer);
  begin
    REGBYTE := 0;
```

```
20300       ADRUNIBUS(ADDRESS, AREA);
20400       case AREA of
20500         ZBUSERR: ERRORS(ZREADBYTE, ZBUSERR, ADDRESS);
20600         ZCPU: BTCPUREAD(ADDRESS, REQBYTE);
20700         ZCORE: BTCORREAD(ADDRESS, REQBYTE);
20800         ZINOUT: BTIOREAD(ADDRESS, REQBYTE)
20900       end
21000     end READBYTE;
21100     procedure export WRITEWORD(ADDRESS:integer; DATA:integer);
21200       begin
21300         ADRUNIBUS(ADDRESS, AREA);
21400         case AREA of
21500           ZBUSERR: ERRORS(ZWRITEWORD, ZBUSERR, ADDRESS);
21600           ZODDADR: ERRORS(ZWRITEWORD, ZODDADR, ADDRESS);
21700           ZCPU: WDCPUWRITE(ADDRESS, DATA);
21800           ZCORE: WDCORWRITE(ADDRESS, DATA);
21900           ZINOUT: WDIOWRITE(ADDRESS, DATA)
22000         end
22100       end WRITEWORD;
22200     procedure export WRITEBYTE(ADDRESS:integer; DATA:integer);
22300       begin
22400         ADRUNIBUS(ADDRESS, AREA);
22500         case AREA of
22600           ZBUSERR: ERRORS(ZWRITEBYTE, ZBUSERR, ADDRESS);
22700           ZCPU: BTCPUWRITE(ADDRESS, DATA);
22800           ZCORE: BTCORWRITE(ADDRESS, DATA);
22900           ZINOUT: BTIOWRITE(ADDRESS, DATA)
23000         end
23100       end WRITEBYTE;
23200     begin %memory\
23300     end MEMORY.
```

```
**END** User KRISHNA [70,105] Job  MEMORY Seq. 2189 Date 18-May-82 12:44:12

**END** User KRISHNA [70,105] Job  MEMORY Seq. 2189 Date 18-May-82 12:44:12

                         * * * L P T S P L  R u n  L o g * * *

12:43:56 LPDAT   [LPTLSJ  LPTSPL version 102(2263) running on LPT101, 18-May-8
12:43:56 LPDAT   [LPTSJS  Starting Job MEMORY, Seq #2189, request created at 1
12:43:56 LPMSG   [LPTSTF  Starting File DSKC:MEMORY.DIP<057>[70,105,OVG]]
12:44:12 LPMSG   [LPTFPF  Finished Printing File DSKC:MEMORY.DIP<057>[70,105,O
12:44:12 LPSUM   Spooler runtime 0 Seconds, 6 KCS, 12 disk reads, 5 pages prin

**END** User KRISHNA [70,105] Job  MEMORY Seq. 2189 Date 18-May-82 12:44:12

**END** User KRISHNA [70,105] Job  MEMORY Seq. 2189 Date 18-May-82 12:44:12

**END** User KRISHNA [70,105] Job  MEMORY Seq. 2189 Date 18-May-82 12:44:12

**END** User KRISHNA [70,105] Job  MEMORY Seq. 2189 Date 18-May-82 12:44:12

**END** User KRISHNA [70,105] Job  MEMORY Seq. 2189 Date 18-May-82 12:44:12

**END** User KRISHNA [70,105] Job  MEMORY Seq. 2189 Date 18-May-82 12:44:12

**END** User KRISHNA [70,105] Job  MEMORY Seq. 2189 Date 18-May-82 12:44:12

**END** User KRISHNA [70,105] Job  MEMORY Seq. 2189 Date 18-May-82 12:44:12

**END** User KRISHNA [70,105] Job  MEMORY Seq. 2189 Date 18-May-82 12:44:12

**END** User KRISHNA [70,105] Job  MEMORY Seq. 2189 Date 18-May-82 12:44:12

**END** User KRISHNA [70,105] Job  MEMORY Seq. 2189 Date 18-May-82 12:44:12
```

START User KRISHNA [70,103] Job UNI Seq. 1596 Date 10-May-82 17:29:50 Mo

START User KRISHNA [70,103] Job UNI Seq. 1596 Date 10-May-82 17:29:50 Mo

```
K   K  RRRR   III   SSSS  H   H  N   N   AAA
K  K   R   R   I   S      H   H  N   N  A   A
K K    R   R   I   S      H   H  NN  N  A   A
KKK    RRRR    I    SSS   HHHHH  N N N  A   A
K  K   R R     I       S  H   H  N  NN  AAAAA
K   K  R  R    I       S  H   H  N   N  A   A
K    K R   R  III  SSSS   H   H  N   N  A   A
```

START User KRISHNA [70,103] Job UNI Seq. 1596 Date 10-May-82 17:29:50 Mo

START User KRISHNA [70,103] Job UNI Seq. 1596 Date 10-May-82 17:29:50 Mo

START User KRISHNA [70,103] Job UNI Seq. 1596 Date 10-May-82 17:29:50 Mo

START User KRISHNA [70,103] Job UNI Seq. 1596 Date 10-May-82 17:29:50 Mo

START User KRISHNA [70,103] Job UNI Seq. 1596 Date 10-May-82 17:29:50 Mo

START User KRISHNA [70,103] Job UNI Seq. 1596 Date 10-May-82 17:29:50 Mo

START User KRISHNA [70,103] Job UNI Seq. 1596 Date 10-May-82 17:29:50 Mo

START User KRISHNA [70,103] Job UNI Seq. 1596 Date 10-May-82 17:29:50 Mo

START User KRISHNA [70,103] Job UNI Seq. 1596 Date 10-May-82 17:29:50 Mo

```
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000001111111111222222222233333333334444444444555555555566666666667777777
1234567890123456789012345678901234567890123456789012345678901234567890123456
```

```
odule UNI options(VERSION = NSIM, DEFAULTCXT)
 include UTILTY, REGDS, GLOBAL, ICSYS;
 var export
   MQVAL, SLVAL, SRVAL, PSVAL, SCVAL: integer;
   REG: array [0 .. 15] of integer;
 type export
   PROCEZR = (ZREADWORD, ZREADBYTE, ZWRITEWORD, ZWRITEBYTE)
   ;
   SUBPROCEZR = (ZCPU, ZCORE, ZINCUI, ZBUSERR, ZODDADR);
 type
   VIEWTYPE = (DECWORD, TDCWORD, BYTE, BITS, CPUMASK,
               IOMASK, CORENASK, REGIST, BYTESELECT,
               PRIORITY);
   WORDTYPE =
     packed record
       case VIEWTYPE of
         DECWORD: (FULLWORD: integer);
         TDCWORD: (DUMMY1TDC: 0 .. 17B;
         LEFTWORD, RIGHTWORD: 0 .. 177777B);
         BYTE: (DUMMYBYTE: 0 .. 17B;
         BYTE3, BYTE2, BYTE1, BYTE0: 0 .. 377B);
         BITS: (BITVAL: packed array [1 .. 36] of boolean);
         CPUMASK: (IGNORE1: 0 .. 3777777B; CPU: 0 .. 1777B;
         GPR: boolean; REGCOUNT: 0 .. 17B;
         PARITY: boolean);
         IOMASK: (IGNORE2: 0 .. 3777777B; IO: 0 .. 17B;
         IGNORE: 0 .. 7777B);
         CORENASK: (IGNORE3: 0 .. 777777B; CORE: 0 .. 3B;
         DUMMY: 0 .. 177777B);
         REGIST: (IGNORE6: 0 .. 3777777B;
         REGBYTE1, REGBYTE0: 0 .. 377B);
         BYTESELECT: (IGNORE7: 0 .. 1777777777B;
         BYTESELECTOR: 0 .. 3B);
         PRIORITY: (IGNORE8: 0 .. 1777777777B;
         PRTYMSK: 0 .. 7; IGNORE9: 0 .. 37B)
     end WORDTYPE;
 const export
   PC = 177740B;
   SP = 177742B;
   MEMSIZE = 14336;
   PS = 177700B;
   SWR = 177702B;
   SLR = 177704B;
   SC = 177734B;
   MQR = 177736B;
   SWR1 = 177703B;
   MQR1 = 177737B;
   SLR1 = 177705B;
 procedure export INITREGS;
   begin
     PSVAL := 0; SLVAL := 0; SCVAL := 0; SRVAL := 0;
     MQVAL := 0;
     for I in 0 to 15
       do REG[I] := 0
   end INITREGS;
 procedure export ERRORS(FROMPROC:PROCEZR;
                         SUBPROC:SUBPROCEZR;
                         ADDRESS:integer);
   var
```

```
    OPERATION: string(9);
    SIZE: string(6);
  begin
    OPERATION := '         ';
    SIZE := '      ';
    SETTRAP(true, TRAP);
    case FROMPROC of


      ZREADWORD:
        begin
          OPERATION := ' reading ';
          SIZE := ' word '
        end;
      ZREADBYTE:
        begin
          OPERATION := ' reading ';
          SIZE := ' byte '
        end;
      ZWRITEWORD:
        begin
          OPERATION := ' writing ';
          SIZE := ' word '
        end;
      ZWRITEBYTE:
        begin
          OPERATION := ' writing ';
          SIZE := ' byte '
        end
    end;
    case SUBPROC of
      ZCPU:
        begin
          SETTRAP(true, BUSTRAP);
          WRITELN(BELL, 'Bustrap for CPU address while ',
                  OPERATION, SIZE, ' at address=', O: 7,
                  ADDRESS)
        end;
      ZCCRF:
        begin
          SETTRAP(true, STRAP);
          WRITELN(BELL, 'Stack over-flow while ',
                  OPERATION, SIZE, ' at address=', O: 7,
                  ADDRESS)
        end;
      ZBUSERR:
        begin
          SETTRAP(true, BUSTRAP);
          WRITELN(BELL, 'Bustrap while ', OPERATION, SIZE,
          'at address=', O: 7, ADDRESS)
        end;
      ZODDADR:
        begin
          SETTRAP(true, BUSTRAP);
          WRITELN(BELL, 'Odd address while', OPERATION,
                  SIZE, ' at address=', O: 7, ADDRESS)
        end
    end
  end ERRORS;
```

```
procedure export SETMASKPRTY(LOCATION,NEWVALUE:integer);
  begin
    if LOCATION = PS
      then SETPRTY(NEWVALUE, PSVAL)
  end SETMASKPRTY;
procedure export WDCPUREAD(ADDRESS:integer;
                           var REQWORD:integer);
  var
    R5: WORDTYPE(CPUMASK);
    R1: WORDTYPE(DECWORD);
  begin
    REQWORD := 0;
    R1.FULLWORD := ADDRESS;
    R5 := WORDTYPE(CPUMASK): R5;
    if R5.GPR
      then GETWORD(REG[R5.REGCOUNT], RIGHT, REQWORD)
      else
      begin
        case ADDRESS of
          PS: GETWORD(PSVAL, RIGHT, REQWORD);
          SLR: GETWORD(SLVAL, RIGHT, REQWORD);
          SC: GETBTE(SCVAL, BYTO, REQWORD);
          MOR: GETWORD(MOVAL, RIGHT, REQWORD);
          SWR: GETWORD(SRVAL, RIGHT, REQWORD);
          otherwise: ERRORS(ZREADWORD, ZCPU, ADDRESS)
        end & CASE\
      end
  end WDCPUREAD;
procedure export BTCPUREAD(ADDRESS:integer;
                           var REQBYTE:integer);
  var
    R5: WORDTYPE(CPUMASK);
    R1: WORDTYPE(DECWORD);
  begin
    REQBYTE := 0;
    R1.FULLWORD := ADDRESS;
    R5 := WORDTYPE(CPUMASK): R5;
    if R5.GPR then
      if R5.PARITY
        then GETBTE(REG[R5.REGCOUNT], BYT1, REQBYTE)
      else GETBTE(REG[R5.REGCOUNT], BYT0, REQBYTE)
    else
    begin
      case ADDRESS of
        PS: GETBTE(PSVAL, BYT0, REQBYTE);
        SLR: GETBTE(SLVAL, BYT0, REQBYTE);
        SLR1: GETBTE(SLVAL, BYT1, REQBYTE);
        SC: GETBTE(SCVAL, BYT0, REQBYTE);
        MOR: GETBTE(MOVAL, BYT0, REQBYTE);
        MOR1: GETBTE(MOVAL, BYT1, REQBYTE);
```

```
        SWB: GETBTE(SRVAL, BYT0, REGBYTE);
        SWB1: GETBTE(SRVAL, BYT1, REGBYTE);
        otherwise: ERRORS(ZREADBYTE, ZCPU, ADDRESS)
      end {CASE}
    end
  end BTCPUREAD;
procedure export WDCPUWRITE(ADDRESS,DATA:integer);
  var
    R5: WORDTYPE(CPUMASK);
    R3: WORDTYPE(BYTE);
    R1: WORDTYPE(DECWORD);
  begin
    R1.FULLWORD := ADDRESS;
    R5 := WORDTYPE(CPUMASK): R1;
    if R5.GPR
      then REG[R5.REGCOUNT] := DATA
    else
    begin
      case ADDRESS of
        PS: PSVAL := DATA;
        SLR: SLVAL := DATA;
        SC: GETBTE(DATA, BYT0, SCVAL);
        MQR: MQVAL := DATA;
        SWR: SRVAL := DATA;
        otherwise: ERRORS(ZWRITEWORD, ZCPU, ADDRESS)
      end
    end
  end WDCPUWRITE;
procedure export BTCPUWRITE(ADDRESS,DATA:integer);
  var
    R5: WORDTYPE(CPUMASK);
    R1: WORDTYPE(DECWORD);
  begin
    R1.FULLWORD := ADDRESS;
    R5 := WORDTYPE(CPUMASK): R1;
    if R5.GPR then
      if R5.PARITY
        then SETBYTE(BYT1, DATA, REG[R5.REGCOUNT])
        else SETBYTE(BYT0, DATA, REG[R5.REGCOUNT])
    else
    begin
      case ADDRESS of
        PS: SETBYTE(BYT0, DATA, PSVAL);
        SLR: SETBYTE(BYT0, DATA, SLVAL);
        SLR1: SETBYTE(BYT1, DATA, SLVAL);
        SC: GETBTE(DATA, BYT0, SCVAL);
        MQR: SETBYTE(BYT0, DATA, MQVAL);
        MQR1: SETBYTE(BYT1, DATA, MQVAL);
        otherwise: ERRORS(ZWRITEBYTE, ZCPU, ADDRESS)
      end
```

```
    end
  end BTCPUWRITE;
Procedure export WDIOREAD(ADDRESS:integer;
                          var REGWORD:integer);
  var
    TEMP: integer;
  begin
    IOREAD(ADDRESS, TEMP);
    GETWORD(TEMP, RIGHT, REGWORD);
  end WDIOREAD;
Procedure export BTIOREAD(ADDRESS:integer;
                          var REGBYTE:integer);
  var
    TEMP: integer;
  begin
    IOREAD(ADDRESS, TEMP);
    GETBYTE(TEMP, BYTO, REGBYTE)
  end BTIOREAD;
Procedure export WDIOWRITE(ADDRESS,DATA:integer);
  var
    TEMP: integer;
  begin
    GETWORD(ADDRESS, RIGHT, TEMP);
    IOWRITE(ADDRESS, TEMP)
  end WDIOWRITE;
Procedure export BTIOWRITE(ADDRESS,DATA:integer);
  var
    TEMP: integer;
    R3: WORDTYPE(BYTE);
    R1: WORDTYPE(DECWORD);
  begin
    GETBYTE(ADDRESS, BYTO, TEMP);
    IOWRITE(ADDRESS, TEMP)
  end BTIOWRITE;
Procedure export REGSET(NEWVALUE,REGNUM:integer);
  begin
    if (0 <= REGNUM) and (REGNUM <= 15)
      then REG[REGNUM] := NEWVALUE
    else WRITELN(BELL,
         ' Illegal register number in procedure regset')
  end REGSET;
Procedure export ADRUNIBUS(ADDRESS:integer;
                           var BLOCK:SUBPROCEZR);
  var
    R1: WORDTYPE(DECWORD);
    R5: WORDTYPE(CPUMASK);
    R6: WORDTYPE(IOMASK);
    R7: WORDTYPE(COREMASK);
  begin
    R1.FULLWORD := ADDRESS;
```

```
23900          R1.FULLWORD := ADDRESS;
24000          R5 := WORDTYPE(CPUMASK): R1;
24100          if not R5.PARITY then
24200          begin
24300            if R5.CPU = 1777B %  IF ADDRESS BELONGS TO CPU  \
24400            then BLOCK := ZCPU
24500            else
24600            begin
24700              R6 := WORDTYPE(IOMASK): R1;
24800              if R6.IO = 17B %  IF ADDRESS BELONGS TO I/O  \
24900                then BLOCK := ZINOUT
25000              else
25100              begin
25200                R7 := WORDTYPE(COREMASK): R1;
25300                if LOGAND(ADDRESS, 600000B) = 0
25400                  then BLOCK := ZCORE
25500                else BLOCK := ZBUSERR
25600              end
25700            end
25800          end
25900          else BLOCK := ZODDADR;
26000        end ADRUNIBUS;
26100      begin %unibus\
26200        PSVAL := 0; SLVAL := 0
26300      end UNI.
```

```
**END** User KRISHNA [70,103]  Job  UNI Seq. 1596 Date 10-May-82 17:30:14 Mo
```


```
**END** User KRISHNA [70,103]  Job  UNI Seq. 1596 Date 10-May-82 17:30:14 Mo
```


```
                    * * * L P T S P L  R u n  L o g * * *

17:29:49 LPDAT  [LPTLSJ  LPTSPL version 102(2263) running on LPT101, 10-May-8
17:29:49 LPDAT  [LPTSJG  Starting Job UNI, seq #1596, request created at 10-M
17:29:51 LPMSG  [LPTSTF  Starting File DSKC:QUV101.LPT<077>[3,3](UNI)]
17:30:13 LPMSG  [LPTFPF  Finished Printing File DSKC:QUV101.LPT<077>[3,3](UNI
17:30:14 LPSUM  Spooler runtime 2 Seconds, 7 KCS, 22 disk reads, 6 pages prin
```


```
**END** User KRISHNA [70,103]  Job  UNI Seq. 1596 Date 10-May-82 17:30:14 Mo

**END** User KRISHNA [70,103]  Job  UNI Seq. 1596 Date 10-May-82 17:30:14 Mo

**END** User KRISHNA [70,103]  Job  UNI Seq. 1596 Date 10-May-82 17:30:14 Mo

**END** User KRISHNA [70,103]  Job  UNI Seq. 1596 Date 10-May-82 17:30:14 Mo

**END** User KRISHNA [70,103]  Job  UNI Seq. 1596 Date 10-May-82 17:30:14 Mo

**END** User KRISHNA [70,103]  Job  UNI Seq. 1596 Date 10-May-82 17:30:14 Mo

**END** User KRISHNA [70,103]  Job  UNI Seq. 1596 Date 10-May-82 17:30:14 Mo

**END** User KRISHNA [70,103]  Job  UNI Seq. 1596 Date 10-May-82 17:30:14 Mo

**END** User KRISHNA [70,103]  Job  UNI Seq. 1596 Date 10-May-82 17:30:14 Mo

**END** User KRISHNA [70,103]  Job  UNI Seq. 1596 Date 10-May-82 17:30:14 Mo

**END** User KRISHNA [70,103]  Job  UNI Seq. 1596 Date 10-May-82 17:30:14 Mo

**END** User KRISHNA [70,103]  Job  UNI Seq. 1596 Date 10-May-82 17:30:14 Mo
```

```
*START* User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:11:21 M
```


```
*START* User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:11:21 M
```


```
K   K  RRRR    III    SSSS  H   H  N   N   AAA
K  K   R   R    I    S      H   H  NN  N  A   A
K K    R   R    I    S      H   H  N N N  A   A
KKK    RRRR     I     SSS   HHHHH  N  NN  A   A
K K    R R      I        S  H   H  N   N  AAAAA
K  K   R  R     I        S  H   H  N   N  A   A
K   K  R   R   III   SSSS   H   H  N   N  A   A
```


```
*START* User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:11:21 M

*START* User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:11:21 M

*START* User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:11:21 M

*START* User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:11:21 M

*START* User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:11:21 M

*START* User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:11:21 M

*START* User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:11:21 M

*START* User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:11:21 M

*START* User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:11:21 M
```


```
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000001111111111222222222233333333334444444444555555555566666666667777777
1234567890123456789012345678901234567890123456789012345678901234567890123456
```

```
module EXEC options(VERSION = NSIM, DEFAULTCXT)
  include MEMORY;
  var export
    SRC, DSTADD, INSTIME, COZSB, SUBREG, OFFSET: integer;
  type export
    ADRTYPE = (ZSRC, ZDSTADD, ZCOZSB, ZSUBREG, ZOFFSET);
    TYPEOPCODE = (JUNK, IOR, IORB, CMP, CMPB, ANDS, ANDB,
                  MUL, DIVX, MOV, MOVB,
                  ADD, SUB, CLE, CLEB, CAL, CLR, CLRB, TST,
                  TSTB, INC, INCB,
                  CAD, CADB, DEC, DECB, CSB, CSBB, COM, COMB
                  , NOTS, NOTB,
                  SHR, SHRB, ESR, ELR, SHL, SHLB, ELL, ENM,
                  CIR, CIRB,
                  CIL, CILB, ECL, ECR, JMP, EXH, DIN, TNE,
                  TGE, TGT, IRA,
                  TEO, TLT, TLE, TFL, TOC, TCC, THI, TMI,
                  TOS, TCS, TLS, SVC,
                  XPR, TRP, SETCC, CLRCC, RES, HLT, WFI, REI
                  , BPT, MCL);
  type
    VIEWTYPE = (DECWORD, TDCWORD, BYTE, BITS, SCNSK,
                PRIORITY);
    WORDTYPE =
      packed record
        case VIEWTYPE of
          DECWORD: (FULLWORD: integer);
          TDCWORD: (DUMMY1TDC: 0 .. 17B;
          LEFTWORD, RIGHTWORD: 0 .. 177777B);
          BYTE: (DUMMYBYTE: 0 .. 17B;
          BYTE3, BYTE2, BYTE1, BYTE0: 0 .. 377B);
          BITS: (BITVAL: packed array [1 .. 36] of boolean);
          SCNSK: (JUNKSC: 0 .. 17777777777B;
          SCNASK: 0 .. 37B);
          PRIORITY: (IGNORE8: 0 .. 777777777B;
          PRTYMASK: 0 .. 7B; IGNORE9: 0 .. 37B)
      end WORDTYPE;
  procedure export INITOPERANDS;
    begin
      SRC := 0; DSTADD := 0; COZSB := 0; SUBREG := 0;
      OFFSET := 0;
      INSTIME := 0
    end INITOPERANDS;
  Procedure export SETADR(NEWVALUE:integer;
    LOCATION:ADRTYPE);
    begin
      case LOCATION of
        ZSRC: SRC := NEWVALUE;
        ZDSTADD: DSTADD := NEWVALUE;
        ZCOZSB: COZSB := NEWVALUE;
        ZSUBREG: SUBREG := NEWVALUE;
        ZOFFSET: OFFSET := NEWVALUE
      end
    end SETADR;
  Procedure export EXECUTE(OPCODE:TYPEOPCODE);
    const
      MASK7 = 200B;
      MASK15 = 100000B;
      MASK31 = 20000000000B;
```

```
  DOUBLEMASK = 377777777778;
  CONWORD = 777778;
  CONBYTE = 1778;
var
  REGWORD, REGBYTE, EXWORD, EXBYTE, COMPWORD, COMPBYTE,
  SHIFTED,
  R, TEMP, CC, SSX, ESRSX, CILSX, CILBSX, ECLSX, ENMSX,

  WANTEDBYTE, WANTEDWORD: integer;
  C, O, Z, S, B, SX, DX: boolean;
  R1: WORDTYPE(DECWORD);
  R2: WORDTYPE(IDCWORD);
  R3: WORDTYPE(BYTE);
  R8: WORDTYPE(BITS);
  R9: WORDTYPE(PRIORITY);
  R10: WORDTYPE(SCMSK);
begin %execute\
  SETFLAG(false, ZHALTFLAG);
  C := BIT(PSVAL, 36);
  O := BIT(PSVAL, 35);
  Z := BIT(PSVAL, 34);
  S := BIT(PSVAL, 33);
  B := BIT(PSVAL, 32);
  INSTIME := 0;
  if (JUNK <= OPCODE) and (OPCODE <= MCL)
    then
    case OPCODE of
      IOR:
        begin
          READWORD(DSTADD, REGWORD);
          R := LOGOR(SRC, REGWORD);
          S := BIT(R, 21);
          O := false;
          Z := (R = 0);
          WRITEWORD(DSTADD, R);
          INSTIME := 22;
        end;
      IORB:
        begin
          READBYTE(DSTADD, REGBYTE);
          R := LOGOR(SRC, REGBYTE);
          S := BIT(R, 29);
          O := false;
          Z := (R = 0);
          WRITEBYTE(DSTADD, R);
          INSTIME := 22
        end;
      CMP:
        begin
          READWORD(DSTADD, R);
          SX := BIT(SRC, 21);
          DX := BIT(R, 21);
          GETWORD(NOTT(R), RIGHT, WANTEDWORD);
          R := SRC + WANTEDWORD + 1;
          S := BIT(R, 21);
          C := not (BIT(R, 2));
          GETWORD(R, RIGHT, WANTEDWORD);
          Z := (WANTEDWORD = 0);
```

```
      O := (SX <> DX) and (SX <> S);
      INSTIME := 24;
    end;
  CMPB:
    begin
      READBYTE(DSTADD, R);
      SX := BIT(SRC, 29);
      DX := BIT(R, 29);
      GETBTE(NOTT(R), BYTO, WANTEDBYTE);
      R := SRC + WANTEDBYTE + 1;
      S := BIT(R, 29);
      C := not (BIT(R, 28));
      GETBTE(R, BYTO, WANTEDBYTE);
      Z := (WANTEDBYTE = 0);
      O := (SX <> DX) and (SX <> S);
      INSTIME := 24;
    end;
  ANDS:
    begin
      READWORD(DSTADD, REQWORD);
      R := LOGAND(SRC, REQWORD);
      S := BIT(R, 21);
      O := false;
      Z := (R = 0);
      INSTIME := 22
    end;
  ANDB:
    begin
      READBYTE(DSTADD, REQBYTE);
      R := LOGAND(SRC, REQBYTE);
      S := BIT(R, 29);
      O := false;
      Z := (R = 0);
      INSTIME := 22
    end;
  MUL:
    begin
      EXTEND(MQVAL, WORDS, EXWORD);
      R := EXWORD;
      EXTEND(SRC, WORDS, EXWORD);
      R := R * EXWORD;
      S := BIT(R, 5);
      GETWORD(R, RIGHT, WANTEDWORD);
      WRITEWORD(MQR, WANTEDWORD);
      GETWORD(R, LEFT, R);
      Z := (R = 0) or (BIT(MQVAL, 21)) and (R =
           177777B);
      C := (MQVAL = 0);
      O := false;
      WRITEWORD(DSTADD, R);
```

```
    INSTIME := 62
  end;
DIVX:
  begin
    if SRC = 0
      then O := true
    else
    begin
      SX := BIT(SRC, 21);
      EXTEND(SRC, WORDs, EXWORD);
      TEMP := EXWORD;
      R := MQVAL;
      READWORD(DSTADD, REGWORD);
      SETWORD(LEFT, REGWORD, R);
      DX := BIT(R, 5);
      if DX
        then R := LOGOR(R, 7400000000B);
      GETWORD(R div TEMP, RIGHT, WANTEDWORD);
      WRITEWORD(MQR, WANTEDWORD);
      GETWORD(R mod TEMP, RIGHT, R);
      C := (MQVAL = 0);
      O := ((SX = DX) <> (BIT(MQVAL, 21) = false))
           ;
      Z := (R = 0);
      S := BIT(R, 21);
      WRITEWORD(DSTADD, R)
    end;
    INSTIME := 90
  end;
MOV:
  begin
    S := BIT(SRC, 21);
    Z := (SRC = 0);
    O := false;
    WRITEWORD(DSTADD, SRC);
    INSTIME := 22
  end;
MOVB:
  begin
    S := BIT(SRC, 29);
    Z := (SRC = 0);
    O := false;
    if REGMODE
      then
    begin
      EXTEND(SRC, BYI, EXBYTE);
      GETWORD(EXBYTE, RIGHT, EXBYTE);
      WRITEWORD(DSTADD, R)
    end
    else WRITEWORD(DSTADD, SRC);
```

```
      INSTIME := 22
    end;
  ADD:
    begin
      READWORD(DSTADD, R);
      DX := BIT(R, 21);
      SX := BIT(SRC, 21);
      R := R + SRC;
      S := BIT(R, 21);
      C := BIT(R, 20);
      GETWORD(R, RIGHT, R);
      Z := (R = 0);
      O := ((SX = DX) and (SX <> S));
      WRITEWORD(DSTADD, R);
      INSTIME := 24;
    end;
  SUB:
    begin
      READWORD(DSTADD, R);
      SX := BIT(SRC, 21);
      DX := BIT(R, 21);
      GETWORD(NOTT(SRC), RIGHT, WANTEDWORD);
      R := R + WANTEDWORD + 1;
      S := BIT(R, 21);
      C := not (BIT(R, 20));
      GETWORD(R, RIGHT, R);
      Z := (R = 0);
      O := ((SX <> DX) and (DX <> S));
      WRITEWORD(DSTADD, R);
      INSTIME := 24;
    end;
  CLR:
    begin
      READWORD(DSTADD, REGWORD);
      R := NOTT(LOGAND(SRC, REGWORD));
      S := BIT(R, 21);
      Z := (R = 0);
      O := false;
      WRITEWORD(DSTADD, R);
      INSTIME := 22
    end;
  CLRB:
    begin
      READBYTE(DSTADD, R);
      R := NOTT(LOGAND(SRC, R));
      S := BIT(R, 29);
      Z := (R = 0);
      O := false;
      WRITEBYTE(DSTADD, R);
      INSTIME := 22
```

```
    end;
  CLR:
    begin
      R := 0;
      WRITEWORD(DSTADD, R);
      S := false;
      Z := true;
      O := false;
      C := false;
      INSTIME := 20
    end;
  CLRB:
    begin
      R := 0;
      WRITEBYTE(DSTADD, R);
      S := false;
      Z := true;
      O := false;
      C := false;
      INSTIME := 20
    end;
  TST:
    begin
      READWORD(DSTADD, R);
      S := BIT(R, 21);
      Z := (R = 0);
      O := false;
      C := false;
      INSTIME := 20
    end;
  CAL:
    begin
      if REGMODE
        then
      begin
        SETTRAP(true, ILLTRAP);
        SETTRAP(true, TRAP);
      end
      else
      begin
        SETMODE(true, ZSPMODE);
        REGSET(REG[1] - 2, 1);
        WRITEWORD(REG[1], REG[SUBREG]);
        REGSET(REG[0], SUBREG);
        REGSET(DSTADD, 0);
        INSTIME := 38
      end;
      return
    end;
  TSTB:
```

```
    begin
      READBYTE(DSTADD, R);
      S := BIT(R, 29);
      Z := (R = 0);
      O := false;
      C := false;
      INSTIME := 20
    end;
  INC:
    begin
      READWORD(DSTADD, REQWORD);
      GETWORD(REQWORD + 1, RIGHT, R);
      Z := (R = 0);
      O := (R = MASK15);
      S := BIT(R, 21);
      WRITEWORD(DSTADD, R);
      INSTIME := 22
    end;
  INCB:
    begin
      READBYTE(DSTADD, REQBYTE);
      GETBTE(REQBYTE + 1, BYTO, R);
      Z := (R = 0);
      O := (R = MASK7);
      S := BIT(R, 29);
      WRITEBYTE(DSTADD, R);
      INSTIME := 22
    end;
  CAD:
    begin
      READWORD(DSTADD, REQWORD);
      if C
        then CC := 1
      else CC := 0;
      R := REQWORD + CC;
      GETWORD(R, RIGHT, R);
      Z := (TEMP = 0);
      O := (TEMP = MASK15) and (CC = 1);
      C := BIT(R, 20);
      S := BIT(TEMP, 21);
      WRITEWORD(DSTADD, TEMP);
      INSTIME := 22
    end;
  CADB:
    begin
      READBYTE(DSTADD, REQBYTE);
      if C
        then CC := 1
      else CC := 0;
      R := REQBYTE + CC;
```

```
      R1.FULLWORD := R;
      R3 := WORDTYPE(BYTE): R1;
      TEMP := R3.BYTE0;
      Z := (TEMP = 0);
      O := (TEMP = MASK7) and (CC = 1);
      C := BIT(R, 28);
      S := BIT(TEMP, 29);
      WRITEBYTE(DSTADD, TEMP);
      INSTIME := 22
    end;
CEC:
    begin
      READWORD(DSTADD, REQWORD);
      R1.FULLWORD := REQWORD - 1;
      R2 := WORDTYPE(TDCWORD): R1;
      R := R2.RIGHTWORD;
      S := BIT(R, 21);
      Z := (R = 0);
      O := (R = CONWORD);
      WRITEWORD(DSTADD, R);
      INSTIME := 22
    end;
CECB:
    begin
      READBYTE(DSTADD, REQBYTE);
      R1.FULLWORD := REQBYTE - 1;
      R3 := WORDTYPE(BYTE): R1;
      R := R3.BYTE0;
      S := BIT(R, 29);
      Z := (R = 0);
      O := (R = CONBYTE);
      WRITEBYTE(DSTADD, R);
      INSTIME := 22
    end;
CSB:
    begin
      READWORD(DSTADD, REQWORD);
      if C
        then CC := 1
      else CC := 0;
      R1.FULLWORD := NOT1(CC);
      R2 := WORDTYPE(TDCWORD): R1;
      R := REQWORD + R2.RIGHTWORD + 1;
      R1.FULLWORD := R;
      R2 := WORDTYPE(TDCWORD): R1;
      TEMP := R2.RIGHTWORD;
      S := BIT(TEMP, 21);
      O := (TEMP = CONWORD) and (CC = 1);
      Z := (TEMP = 0);
      C := not (BIT(R, 28));
```

```
      WRITEWORD(DSTADD, TEMP);
      INSTIME := 22
    end;
  CSBB:
    begin
      READBYTE(DSTADD, REGBYTE);
      if C
        then CC := 1
        else CC := 0;
      R1.FULLWORD := NCTI(CC);
      R3 := WORDTYPE(BYTE): R1;
      R := REGBYTE + R3.BYTE0 + 1;
      R1.FULLWORD := R;
      R3 := WORDTYPE(BYTE): R1;
      TEMP := R3.BYTE0;
      S := BIT(TEMP, 29);
      O := (TEMP = COMBYTE) and (CC = 1);
      Z := (TEMP = 0);
      C := not (BIT(R, 28));
      WRITEWORD(DSTADD, TEMP);
      INSTIME := 22
    end;
  COM:
    begin
      READWORD(DSTADD, REGWORD);
      R1.FULLWORD := NCTI(REGWORD) + 1;
      R2 := WORDTYPE(TDCWORD): R1;
      R := R2.RIGHTWORD;
      Z := (R = 0);
      O := (R = MASK15);
      C := (R <> 0);
      S := BIT(R, 21);
      WRITEWORD(DSTADD, R);
      INSTIME := 22
    end;
  COMB:
    begin
      READBYTE(DSTADD, REGBYTE);
      R1.FULLWORD := NCTI(REGBYTE) + 1;
      R3 := WORDTYPE(BYTE): R1;
      R := R3.BYTE0;
      Z := (R = 0);
      O := (R = MASK7);
      C := (R <> 0);
      S := BIT(R, 29);
      WRITEWORD(DSTADD, R);
      INSTIME := 22
    end;
  NOTS:
    begin
```

```
      READWORD(DSTADD, REQWORD);
      GETWORD(NOTT(REQWORD), RIGHT, R);
      S := BIT(R, 21);
      Z := (R = 0);
      O := false;
      C := true;
      WRITEWORD(DSTADD, R);
      INSTIME := 20
    end;
  NOTB:
    begin
      READBYTE(DSTADD, REQBYTE);
      GETBYTE(NOTT(REQBYTE), BYTO, R);
      S := BIT(R, 29);
      Z := (R = 0);
      O := false;
      C := true;
      WRITEBYTE(DSTADD, R);
      INSTIME := 20
    end;
  SHR:
    begin
      READWORD(DSTADD, R);
      C := BIT(R, 36);
      S := BIT(R, 21);
      RSHIFT(R, 1, SHIFTED);
      R := LOGOR(SHIFTED, LOGAND(R, MASK15));
      Z := (R = 0);
      O := (S <> C);
      WRITEWORD(DSTADD, R);
      INSTIME := 22
    end;
  SHRB:
    begin
      READBYTE(DSTADD, R);
      C := BIT(R, 36);
      S := BIT(R, 29);
      RSHIFT(R, 1, SHIFTED);
      R := LOGOR(SHIFTED, LOGAND(R, MASK7));
      Z := (R = 0);
      O := (S <> C);
      WRITEBYTE(DSTADD, R);
      INSTIME := 22
    end;
  FSR:
    begin
      R := MOVAL;
      READWORD(DSTADD, REQWORD);
      SETWORD(LEFT, REQWORD, R);
      S := BIT(R, 5);
```

```
        ESRSX := LOGAND(R, MASK31);
        INSTIME := 02 * SCVAL + 22;
        while SCVAL > 0
              do
        begin
          WRITEWORD(SC, SCVAL - 1);
          C := BIT(R, 36);
          RSHIFT(R, 1, SHIFTED);
          R := LOGOR(SHIFTED, ESRSX)
        end;
        GETWORD(R, RIGHT, WANTEDWORD);
        WRITEWORD(MQR, WANTEDWORD);
        GETWORD(R, LEFT, R);
        O := false;
        Z := (R = 0) or (BIT(MQVAL, 21)) and (R =
             177777B);
        WRITEWORD(DSTADD, R)
      end;
    FLR:
      begin
        WRITEWORD(SC, LOGAND(SCVAL, 37B));
        R := MQVAL;
        READWORD(DSTADD, REQWORD);
        SETWORD(LEFT, REQWORD, R);
        if SCVAL > 0
          then C := BIT(R, SCVAL - 1);
        INSTIME := 02 * SCVAL + 20;
        RSHIFT(R, SCVAL, SHIFTED);
        R := SHIFTED;
        GETWORD(R, RIGHT, WANTEDWORD);
        WRITEWORD(MQR, WANTEDWORD);
        GETWORD(R, LEFT, R);
        O := false;
        S := BIT(R, 21);
        Z := (R = 0) or (BIT(MQVAL, 21)) and (R =
             177777B);
        WRITEWORD(DSTADD, R)
      end;
    SHI:
      begin
        READWORD(DSTADD, R);
        C := BIT(R, 21);
        LSHIFT(R, 1, SHIFTED);
        GETWORD(SHIFTED, RIGHT, R);
        S := BIT(R, 21);
        O := (S <> C);
        Z := (R = 0);
        WRITEWORD(DSTADD, R);
        INSTIME := 22
      end;
```

```
SHLB:
  begin
    READBYTE(DSTADD, R);
    C := BIT(R, 29);
    LSHIFT(R, 1, SHIFTED);
    GETBYTE(SHIFTED, BYTE0, R);
    S := BIT(R, 29);
    O := (S <> C);
    Z := (R = 0);
    WRITEBYTE(DSTADD, R);
    INSTIME := 22
  end;
FLL:
  begin
    WRITEWORD(SC, LOGAND(SCVAL, 37B));
    R := MQVAL;
    READWORD(DSTADD, REGWORD);
    SETWORD(LEFT, REGWORD, R);
    INSTIME := 02 * SCVAL + 22;
    if SCVAL > 0
      then C := BIT(R, 32 - SCVAL);
    LSHIFT(R, SCVAL, SHIFTED);
    O := false;
    GETWORD(SHIFTED, LEFT, R);
    GETWORD(SHIFTED, RIGHT, WANTEDWORD);
    WRITEWORD(MQR, WANTEDWORD);
    S := BIT(R, 21);
    Z := (R = 0) or (BIT(MQVAL, 21)) and (R =
         177777B);
    WRITEWORD(DSTADD, R)
  end;
ENM:
  begin
    R := MQVAL;
    SETWORD(LEFT, REGWORD, R);
    ENMSX := 0;
    while (BIT(R, 5) = BIT(R, 6)) and (ENMSX < 31)
          do
    begin
      ENMSX := ENMSX + 1;
      LSHIFT(R, 1, SHIFTED);
      R := SHIFTED
    end;
    GETWORD(R, RIGHT, WANTEDWORD);
    WRITEWORD(MQR, WANTEDWORD);
    GETWORD(R, LEFT, R);
    C := (MQVAL = 0);
    O := false;
    Z := (R = 0);
    S := BIT(R, 21);
```

```
      COMPLEMENT(ENMSX, BYT, COMPBYTE);
      WRITEWORD(SC, COMPBYTE);
      WRITEWORD(DSTADD, R);
      INSTIME := 92 * ENMSX + 20
    end;
  CIL:
    begin
      READWORD(DSTADD, R);
      if C
        then CILSX := 1
        else CILSX := 0;
      C := BIT(R, 21);
      LSHIFT(R, 1, SHIFTED);
      GETWORD(SHIFTED, RIGHT, R);
      R := LOGOR(R, CILSX);
      Z := (R = 0);
      S := BIT(R, 21);
      O := (S <> C);
      WRITEWORD(DSTADD, R);
      INSTIME := 22
    end;
  CILB:
    begin
      READBYTE(DSTADD, R);
      if C
        then CILBSX := 1
        else CILBSX := 0;
      C := BIT(R, 29);
      LSHIFT(R, 1, SHIFTED);
      GETBIE(SHIFTED, BYTO, R);
      R := LOGOR(R, CILBSX);
      S := BIT(R, 29);
      Z := (R = 0);
      O := (S <> C);
      WRITEBYTE(DSTADD, R);
      INSTIME := 22
    end;
  CIR:
    begin
      READWORD(DSTADD, R);
      S := C;
      C := BIT(R, 36);
      RSHIFT(R, 1, SHIFTED);
      R := SHIFTED;
      if S
        then R := LOGOR(R, MASK15);
      O := (S <> C);
      Z := (R = 0);
      WRITEWORD(DSTADD, R);
      INSTIME := 22
```

```
    end;
  CIRB:
    begin
      READBYTE(DSTADD, R);
      S := C;
      C := BIT(R, 36);
      RSHIFT(R, 1, SHIFTED);
      R := SHIFTED;
      if S
        then R := LOGOR(R, MASK7);
      O := (S <> C);
      Z := (R = 0);
      WRITEBYTE(DSTADD, R);
      INSTIME := 22
    end;
  ECL:
    begin
      R1.FULLWORD := SCVAL;
      R10 := WORDTYPE(SCMSK); R1;
      WRITEWORD(SC, LOGAND(SCVAL, 37B));
      R := MQVAL;
      READWORD(DSTADD, REGWORD);
      SETWORD(LEFT, REGWORD, R);
      INSTIME := 02 * SCVAL + 22;
      while SCVAL > 0
            do
      begin
        WRITEWORD(SC, SCVAL - 1);
        if C
          then ECLSX := 1
        else ECLSX := 0;
        C := BIT(R, 5);
        LSHIFT(R, 1, SHIFTED);
        R := LOGOR(SHIFTED, ECLSX)
      end;
      S := BIT(R, 5);
      O := false;
      GETWORD(R, RIGHT, WANTEDWORD);
      WRITEWORD(MQR, WANTEDWORD);
      R := R2.LEFTWORD;
      Z := (R = 0) or (BIT(MQVAL, 21)) and (R =
          177777B);
      WRITEWORD(DSTADD, R)
    end;
  FCR:
    begin
      WRITEWORD(SC, LOGAND(SCVAL, 37B));
      R := MQVAL;
      READWORD(DSTADD, REGWORD);
      SETWORD(LEFT, REGWORD, R);
```

```
      INSTIME := 02 * SCVAL + 22;
      while SCVAL > 0
            do
      begin
        WRITEWORD(SC, SCVAL - 1);
        S := C;
        C := BIT(R, 36);
        RSHIFT(R, 1, SHIFTED);
        R := SHIFTED;
        if S
          then R := LOGOR(R, MASK31)
      end;
      GETWORD(R, RIGHT, WANTEDWORD);
      WRITEWORD(MOR, WANTEDWORD);
      O := false;
      Z := (R = 0) or (BIT(MOVAL, 21)) and (R =
           177777B);
      WRITEWORD(DSTADD, R)
    end;
JMP:
    begin
      if REGMODE
          then
      begin
        SETTRAP(true, TRAP);
        SETTRAP(true, ILLTRAP);
      end
      else REGSET(DSTADD, 0);
      INSTIME := 20;
      return
    end;
EXB:
    begin
      READWORD(DSTADD, R);
      GETBYTE(R, BYTO, TEMP);
      R := R3.BYTE1;
      S := BIT(R, 29);
      Z := (R = 0);
      O := false;
      C := false;
      SETBYTE(BYT1, TEMP, R);
      WRITEWORD(DSTADD, R);
      INSTIME := 20
    end;
CTN:
    begin
      GETBYTE(SCVAL - 1, BYTO, WANTEDBYTE);
      WRITEWORD(SC, WANTEDBYTE);
      if SCVAL = 0
          then
```

```
    begin
      DSTADD := REG[0];
      INSTIME := 18
    end
    else
    begin
      EXTEND(OFFSET, BYT, EXBYTE);
      GETWORD(REG[0] + EXBYTE * 2, RIGHT,
              WANTEDWORD);
      DSTADD := WANTEDWORD;
      INSTIME := 26
    end;
    REGSET(DSTADD, 0);
    return
  end;
INE:
  begin
    if Z
      then
    begin
      DSTADD := REG[0];
      INSTIME := 18
    end
    else
    begin
      EXTEND(OFFSET, BYT, EXBYTE);
      GETWORD(REG[0] + EXBYTE * 2, RIGHT,
              WANTEDWORD);
      DSTADD := WANTEDWORD;
      INSTIME := 26
    end;
    REGSET(DSTADD, 0);
    return
  end;
TGE:
  begin
    if S or O
      then
    begin
      DSTADD := REG[0];
      INSTIME := 18
    end
    else
    begin
      EXTEND(OFFSET, BYT, EXBYTE);
      GETWORD(REG[0] + EXBYTE * 2, RIGHT,
              WANTEDWORD);
      DSTADD := WANTEDWORD;
      INSTIME := 26
    end;
```

```
    REGSET(DSTADD, 2);
    return
  end;
IGT:
  begin
    if Z or O or S
      then
    begin
      DSTADD := REG[0];
      INSTIME := 18
    end
    else
    begin
      EXTEND(OFFSET, BYT, EXBYTE);
      GETWORD(REG[0] + EXBYTE * 2, RIGHT,
              WANTEDWORD);
      DSTADD := WANTEDWORD;
      INSTIME := 26
    end;
    REGSET(DSTADD, 0);
    return
  end;
IRA:
  begin
    EXTEND(OFFSET, BYT, EXBYTE);
    GETWORD(REG[0] + EXBYTE * 2, RIGHT, WANTEDWORD
            );
    DSTADD := WANTEDWORD;
    REGSET(DSTADD, 0);
    INSTIME := 26;
    return
  end;
IEQ:
  begin
    if not (Z)
      then
    begin
      DSTADD := REG[0];
      INSTIME := 18
    end
    else
    begin
      EXTEND(OFFSET, BYT, EXBYTE);
      GETWORD(REG[0] + EXBYTE * 2, RIGHT,
              WANTEDWORD);
      DSTADD := WANTEDWORD;
      INSTIME := 26
    end;
    REGSET(DSTADD, 0);
    return
```

```
  end;
JLT:
  begin
    if not (S or O)
      then
    begin
      DSTADD := REG[0];
      INSTIME := 18
    end
    else
    begin
      EXTEND(OFFSET, BYT, EXBYTE);
      GETWORD(REG[0] + EXBYTE * 2, RIGHT,
              WANTEDWORD);
      DSTADD := WANTEDWORD;

      INSTIME := 26
    end;
    REGSET(DSTADD, 0);
    return
  end;
JLE:
  begin
    if (not (Z)) or (S or O)
      then
    begin
      DSTADD := REG[0];
      INSTIME := 18
    end
    else
    begin
      EXTEND(OFFSET, BYT, EXBYTE);
      GETWORD(REG[0] + EXBYTE * 2, RIGHT,
              WANTEDWORD);
      DSTADD := WANTEDWORD;
      INSTIME := 26
    end;
    REGSET(DSTADD, 0);
    return
  end;
JOC:
  begin
    if O
      then
    begin
      DSTADD := REG[0];
      INSTIME := 18
    end
    else
```

```
        begin
          EXTEND(OFFSET, BYT, EXBYTE);
          GETWORD(REG[0] + EXBYTE * 2, RIGHT,
                  WANTEDWORD);
          DSTADD := WANTEDWORD;
          INSTIME := 26
        end;
        REGSET(DSTADD, 0);
        return
      end;
    TPL:
      begin
        if S
          then
        begin
          DSTADD := REG[0];
          INSTIME := 18
        end
        else
        begin
          EXTEND(OFFSET, BYT, EXBYTE);
          GETWORD(REG[0] + EXBYTE * 2, RIGHT,
                  WANTEDWORD);
          DSTADD := WANTEDWORD;
          INSTIME := 26
        end;
        REGSET(DSTADD, 0);
        return
      end;
    JCC:
      begin
        if C
          then
        begin
          DSTADD := REG[0];
          INSTIME := 18
        end
        else
        begin
          EXTEND(OFFSET, BYT, EXBYTE);
          GETWORD(REG[0] + EXBYTE * 2, RIGHT,
                  WANTEDWORD);
          DSTADD := WANTEDWORD;
          INSTIME := 26

        end;
        REGSET(DSTADD, 0);
        return
      end;
    THI:
```

```
    begin
      if C or Z
        then
      begin
        DSTADD := REG[0];
        INSTIME := 18
      end
      else
      begin
        EXTEND(OFFSET, BYT, EXBYTE);
        GETWORD(REG[0] + EXBYTE * 2, RIGHT,
                WANTEDWORD);
        DSTADD := WANTEDWORD;
        INSTIME := 26
      end;
      REGSET(DSTADD, 0);
      return
    end;
  IMI:
    begin
      if not (S)
        then
      begin
        DSTADD := REG[0];
        INSTIME := 18
      end
      else
      begin
        EXTEND(OFFSET, BYT, EXBYTE);
        GETWORD(REG[0] + EXBYTE * 2, RIGHT,
                WANTEDWORD);
        DSTADD := WANTEDWORD;
        INSTIME := 26
      end;
      REGSET(DSTADD, 0);
      return
    end;
  IDS:
    begin
      if not (D)
        then
      begin
        DSTADD := REG[0];
        INSTIME := 18
      end
      else
      begin
        EXTEND(OFFSET, BYT, EXBYTE);
        GETWORD(REG[0] + EXBYTE * 2, RIGHT,
                WANTEDWORD);
```

```
      DSTADD := WANTEDWORD;
      INSTIME := 26
    end;
    REGSET(DSTADD, 0);
    return
  end;
TCS
;
  begin
    if not (C)
      then
    begin
      DSTADD := REG[0];
      INSTIME := 19
    end
    else
    begin
      EXTEND(OFFSET, BYT, EXBYTE);
      GETWORD(REG[0] + EXBYTE * 2, RIGHT,
              WANTEDWORD);
      DSTADD := WANTEDWORD;
      INSTIME := 26
    end;
    REGSET(DSTADD, 0);
    return
  end;
TLS:
  begin
    if not (C) and not (Z)

      then
    begin
      DSTADD := REG[0];
      INSTIME := 18
    end
    else
    begin
      EXTEND(OFFSET, BYT, EXBYTE);
      GETWORD(REG[0] + EXBYTE * 2, RIGHT,
              WANTEDWORD);
      DSTADD := WANTEDWORD;
      INSTIME := 26
    end;
    REGSET(DSTADD, 0);
    return
  end;
SVC:
  begin
    SETMODE(true, ZSEMODE);
```

```
      SERTRAP(IOTLOCATION);
      return
    end;
XPR:
    begin
      SETMODE(true, ZSEMODE);
      SERTRAP(EMTLOCATION);
      return
    end;
TRP:
    begin
      SETMODE(true, ZSEMODE);
      SERTRAP(TRPLOCATION);
      return
    end;
SETCC;
    begin
      C := C or BIT(CCZSB, 36);
      O := O or BIT(CCZSB, 35);
      Z := Z or BIT(CCZSB, 34);
      S := S or BIT(CCZSB, 33);
      B := B or BIT(CCZSB, 32);
      INSTIME := 18
    end;
CLRCC;
    begin
      C := C or not (BIT(CCZSB, 36));
      O := O or not (BIT(CCZSB, 35));
      Z := Z or not (BIT(CCZSB, 34));
      S := S or not (BIT(CCZSB, 33));
      B := B or not (BIT(CCZSB, 32));
      INSTIME := 18
    end;
RES;
    begin
      REGSET(REG[SUBREG], 0);
      READWORD(REG[1], REGWORD);
      REGSET(REGWORD, SUBREG);
      REGSET(REG[1] + 2, 1);
      INSTIME := 38;
      return
    end;
FLT;
    begin
      SETFLAG(true, ZHALTFLAG);
      SETMODE(false, ZRUNMODE);
      INSTIME := 18;
      return
    end;
WFI;
```

```
        begin
          SETMODE(true, ZRUNMODE);
          INSTIME := 18;
          return
        end;
      REI:
        begin
          READWORD(REG[1], REGWORD);
          REGSET(REGWORD, 0);
          REGSET(REG[1] + 2, 1);
          READWORD(REG[1], REGWORD);
          WRITEWORD(PS, REGWORD);
          REGSET(REG[1] + 2, 1);
          INSTIME := 50;
          return
        end;
      BPT:
        begin
          SETMODE(true, ZSPECIAL);
          SERTRAP(BPTLOCATION);
          return
        end;
      MCL:
        begin
          MCLREALS;
          INSTIME := 200000;
          return
        end
    end { IF OPCODE IS NOT IN MNEMIC }\
  else
  begin
    WRITELN(' ?? ILLEGAL OPCODE IN EXECUTE ROUTINE ')
    #FINISH\
  end;
  if TRAPFLAG
    then return ;
  R1.FULLWORD := PSVAL;
  R9 := WORDTYPE(PRIORITY): R1;
  WRITEWORD(PS, R9.PRTYMASK);
  R1.FULLWORD := R9.PRTYMASK;
  R8 := WORDTYPE(BITS): R1;
  for J in 1 to 5
    do
    for I in 2 to 36
      do R8.BITVAL[I - 1] := R8.BITVAL[I];
  R8.BITVAL[32] := B;
  R8.BITVAL[33] := S;
  R8.BITVAL[34] := Z;
  R8.BITVAL[35] := O;
```

```
04500          R8.BITVAL[36] := C;
04600          R1 := WORDTYPE(DECWORD): R8;
04700          WRITEWORD(PS, R1.FULLWORD);
04800          return
04900        end EXECUTE;
05000      begin
05100        INSTIME := 0;
05200        SRC := 0; DSTADD := 0
05300      end EXEC.
```

```
**END** User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:13:12 M

**END** User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:13:12 M

                    * * * L P T S P L  R u n  L o g * * *

18:11:15 LPDAT  [LPTLSJ  LPTSPL version 102(2263) running on LPT101, 10-May-8
18:11:15 LPDAT  [LPTSJS  Starting Job EXEC, Seq #1650, request created at 10-
18:11:22 LPMSG  [LPTSTF  Starting File DSKC:QVO101.LPT<077>[3,3](EXEC)]
18:13:11 LPMSG  [LPTFPF  Finished Printing File DSKC:QVO101.LPT<077>[3,3](EXE
18:13:12 LPSUM  Spooler runtime 2 Seconds, 18 KCS, 65 disk-reads, 24 pages pr

**END** User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:13:12 M

**END** User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:13:12 M

**END** User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:13:12 M

**END** User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:13:12 M

**END** User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:13:12 M

**END** User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:13:12

**END** User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:13:12

**END** User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:13:12

**END** User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:13:12

**END** User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:13:12

**END** User KRISHNA [70,103] Job EXEC Seq. 1650 Date 10-May-82 18:13:12
```

B I B L I O G R A P H Y

1. Parnas D.L.: On the criteria to be used in decomposing systems into modules. 1972 Dec. CACM 15,12; pp 1053-1058

2. Parnas D.L.: A technique for Software Module Specification With Examples. 1972 May. CACM 15,5; pp 330-336

3. Parnas D.L.: Information Distribution aspects of design methodology. 1971. Technical report of Computer Science, C.M.U - Pittsburgh. Also in IFIP Congress 1971.

4. Parnas D.L.: Software Engineering methods for the multi-person Construction of multi-level programs. 1974 Sept. Lecture Notes in Computer Science, Vol 23.

5. Parnas D.L.: Designing software for case of extension and contraction. 1979 March. IEEE Software Engineering, SE-5, No. 2; pp 128-138

6. Ben-Ari, A. Yahudi: A methodology for modular use of Ada. 1981 Sept. SIGPLAN Notices 16,12; pp 22-27

7. Wirth N.: On the composition of well structured programs. 1974 Dec. Computing Surveys.

8. Bergland G.D.: A guided tour of program design methodologies. 1981 Oct. Computer

9. Joshua Turner, Penn Mutual Life: The Structure of modular programs. 1980 CACM 23,5 pp 272-277

10. Trost D.: Psychology of program design. 1975 May. Datamation 21,5 pp 137-138

11. Yourdon E.: Technique of program structure and design. 1975 Prentice Hall, Englewood Cliffs.

12. Aron J.D.: The program development process. 1974. Part 1 The Individual Programmer Addison-Wesley, Reading, Mass.

13. Dahl O.J., Dijkastra E.W. and Hoore C.A.R.: Structured programming 1972 Academic Press, London.

14. Dijkstra E.W.: The Humble Programmer. 1972 CACM Turing Award Lecture.

15. V.R. Prasad: Report on the concurrent programming language CCNPASCAL. 1978 Oct. Technical Report no. 28, NCSDCT.

16. Mathai Joseph, Srinivas (TIFR): Structured Programming using decision tables. Journal of C.S.I. 1973.

17. V.R. Prasad: Program document No.2 Simulator for TDC 316, BCPL Version. 1975 (Revised: October, 1979).

18. Wirth N.: Program development by stepwise refinement 1971 April. CACM 14,4 pp 221-228

19. Herbert Maisel and Giuliana Gnugnoti: Simulation of Discrete Stockastic Systems. 1972. SRA edition.

20. LuCas H.C.Jr: Performance evaluation and monitoring 1971 ACM Computing Surveys 3(3) pp 79-91

21. Programming Methodology: A collection of Articles by Members of IFIP W6 2,3. Edited by David Gries; Springerverlag.

22. Software Engineering - Concepts and Techniques Edited by J.M. Buxton et al Petrocelli/charter 1976.

23. Gries D. et al : Some techniques used in the ALCOR ILLINOIS 7090. 1965, CACM 8; pp 496-900 .

24. Poole P.C. and Waile W.M.: Machine independent Software. 1969 Oct. Proceedings of ACM, second symposium on O.S. Principles, Princeton, N.Y.

25. Waite W.M.: The mobile programming system: STAGE 2. 1970 CACM 13, 15.

26. Kallol K. Bagchi et al: The use of modular simulator in the design and architectural studies of micro computer system. Proceedings of the 6th Anmual Convention of C.S.I. 81, 120225, New Delhi, India.