intellitech

The Intelligent Use of Technology

(2)

SIMULATION OF THE SBP 9989 MICROPROCESSOR USING THE CAE TOOL N.mPc ON A VAX 11/780

INT-83-651

SIMULATION OF THE SBP 9989 MICROPROCESSOR USING THE CAE TOOL N.mPc ON A VAX 11/780

SEPTEMBER 1984

Prepared By: Max Streit

Approved By: Dr. S.A. Mahmoud
Dr. C. Laferriere

INTELLITECH CANADA LIMITED

352 MacLaren Street
Ottawa, Ontario
K2P 0M6

Government of Canada / Gouvernement du Canada

Department of Communications

DOC CONTRACTOR REPORT DOC-CR-SP -84-023

DEPARTMENT OF COMMUNICATIONS - OTTAWA - CANADA

SPACE PROGRAM

TITLE: Simulation of the SBP 9989 Microprocessor Using the Computer Aided Engineering Tool N.mPc on a VAX 11/780

AUTHOR(S): Max Streit
INTELLITECH CANADA LIMITED
352 MacLaren Street
Ottawa, Ontario
K2P 0M6

ISSUED BY CONTRACTOR AS REPORT NO: INT-83-65i

PREPARED BY: Max Streit

DEPARTMENT OF SUPPLY AND SERVICES CONTRACT NO: OER 83-05075

DOC SCIENTIFIC AUTHORITY: Michel Savoie
COMMUNICATIONS RESEARCH CENTRE
Ottawa, Ontario

CLASSIFICATION: unclassified

DATE: SEPTEMBER 1984

SUMMARY

This report describes the simulation of the Texas Instrument SBP 9989 microprocessor using the computer aided engineering design and development tool N.mPc, developed by Case Western University. The simulation was developed and run on the VAX 11/780 System of the Analysis and Simulation Laboratory, Communications Research Centre. The simulation includes a hardware description, a command file for the programmable assembler and a command file for the programmable linking/loader. A test program is also included. Taken together, these components constitute a software implementation of the SBP 9989 as well as a complete software development system for the SBP 9989 on the VAX 11/780. The architecture of SBP 9989 is discussed, its implementation in N.mPc and a test simulation are described. Comments based on the experience gained during the hardware and software simulation of the SBP 9989 processor are provided. This report describes part of the work done under contract OER83-05075 for the Communications Research Centre of the Department of Communications, Government of Canada.

FOREWORD

This report describes work involving processor hardware description and simulation using the Computer Aided Engineering (CAE) tool N.mPc which runs on the VAX 11/780 of the Analysis and Simulation Laboratory, CRC. As in all new fields of endeavour, CAE design activities introduce new concepts, hence a new vocabulary. particularly confusing at first is the general tendency of re-using terminology which may have a different meaning in other engineering fields. The purpose of this foreword is, therefore, to provide a common point of reference with regard to the various terms used in the report. This will be achieved by briefly describing the use of the CAE tool N.mPc.

Traditionally, microcomputer based products have been designed in the following fashion:

1. The necessary hardware components are built. This usually includes the microcomputer itself which is build using a microprocessor and other peripheral components.
2. The software programs are written for the target machine.
3. Software and hardware components are integrated and tested. Very frequently, the software is produced on a host machine using a cross development package (if available).

The development process usually involves many time consuming and costly iterations. A CAE tool such as N.mPc improves the situation by providing a simulation environment which is suitable for testing many design alternatives in a short period of time. The implications of using N.mPc are as follows:

1. It is no longer necessary to build the hardware components at the beginning of the design work. Instead, N.mPc provides what amounts to a micro-programmable, register transfer level machine which can be programmed to emulate the target hardware completely. In other words, a designer working on a VAX host for example, could create a VAX executable program which, when run, would emulate the target hardware.

2. N.mPc provides a cross development package for the software to be written. The cross development facilities are, however, totally programmable. Programming the cross development facilities of N.mPc is part of the preparatory work which needs be done only once for a given type of hardware.

3. Facilities exist in N.mPc to take the user written, application software produced by the cross development package and to incorporate it into the hardware emulation module. The result thus becomes the execution of the application software by the target hardware which in turn is an emulation run by the host computer.

4. The rationale for using a tool such as N.mPc is that programmability implies flexibility. Given that a base exists, i.e. most of the hardware emulation is available as well as the cross development package, a designer can alter the design parameters with ease and test various alternatives without committing to any hardware choice.

N.mPc introduces new activities and redefines some traditional concepts. The hardware building stage which is to result in a host executable emulation involves programming the register transfer level machine provided by N.mPc. This programming is done in the ISP'

language and is referred to as "implementing" the hardware or preparing the hardware description. Similarly, the programming of the cross-development package involves the preparation of various command files and is considered part of the "implementation" of the target hardware.

A complete N.mPc simulation is the execution of a complete simulation module which includes the hardware emulation and the application programs which have been developed by the cross development package. The fact that the hardware is also simulated may lead to confusion. Therefore, the hardware simulation is usually referred to as emulation whereas the term simulation is reserved for the execution of user programs on the emulated hardware.

This brief foreword, however, falls short of explaining the complexities of a system like N.mPc. Its main purpose was to warn prospective readers that some terms such as simulation, emulation and implementation have a very special meaning in the N.mPc context. The interested reader will find many references to work reporting applications of N.mPc in the field of processor design. A list of references is included at the end of this report.

TABLE OF CONTENTS

TABLE OF FIGURES

1.0 INTRODUCTION

1.1 Background

The Texas Instruments TMS 9900 was one of the first 16-bit microprocessors. Introduced in 1977, the TMS 9900 is produced using NMOS technology. Its "memory-to-memory" architecture uses blocks of external memory, designated as "workspace", instead of internal hardware registers; this speeds up context switches associated with interrrupts, subroutine calls, etc. The TMS 9900 has 69 instructions and 64K bytes of memory addressed as 32K 16-bit words.

The SBP 9989 is an upgraded military version of the TMS 9900 microprocessor. The SBP9989 differs from its civilian version mainly in the use of Integrated Injection Logic (I^2L) technology which provides better reliability in extreme temperature conditions and in radiation environments. The instruction set of the SBP 9989 includes all of the TMS 9900 instructions plus four additional instructions which improve performance in numerical applications and multiprocessor configurations.

Because I^2L based components are inherently radiation hard the SBP 9989 not only conforms to military standards (MIL 883B) but also is space qualified. It is the main processing unit in the "Attitude and Orbit Control System (AOCS) of the European Space Agency (ESA) Large Satellite (LSAT) developed by British Aerospace.

Initially the Ferranti F100L processor had been chosen for the LSAT AOCS. The need for additional processing capacity led to the decision to switch to the SBP 9989 processor for the LSAT program. An independent investigation [2] carried out by Intellitech Canada Limited in the fall of 1983, for the Analysis and Simulation Laboratory of the Communications Research Centre, came to the same conclusions with regard, to the relative performance of the two processors.

1.2 Simulation - The Modern Approach To Microcomputer System Design

The traditional approach to microcomputer system design usually tries to build a prototype of the system to be developed. This prototype will consist of a number of hardware boards assembled using wire wrap techniques. This approach makes design changes difficult and becomes costly for the case of large, complex systems (e.g. multiprocessor systems). The designer also faces problems related to the implementation of his hardware design while he is, at the same time, trying to debug the system software. The traditional approach makes it difficult to determine whether the software design or the hardware design are responsible for certain failures and/or inadequacies in the system.

The modern approach attempts first to simulate a design on a host computer using a "computer aided engineering (CAE) tool such as N.mPc. This allows easy development and verification of both hardware and software designs because the simulation is easy to change. Complex systems are handled by using modular simulation software. Once the new design is verified, the hardware is built and the software that previously ran on the simulated hardware can now be reassembled and loaded to be run on the actual hardware.

Figure 1.1 shows the main components of N.mPc, a CAE tool capable of simulating complex systems by modelling the hardware at the register transfer level. The user defines the hardware modules of his system (in ISP´ source) and their configuration (Topology) and then assembles, links and loads the microprocessor software to be run. The N.mPc system processes all this information and forms a simulation that is easily controlled by the user. Changes and reconfiguration of the hardware modules is a matter of executing a few commands. Fault insertion,

mathematical and graphical post processing of the simulation results are also features of N.mPc. The N.mPc system used for this work ran on a VAX 11/780 under the VMS operating system.

This work started with a description of the TMS 9900 from the N.mPc library. Although this 9900 module did compile correctly, a fair amount of work had to be done to check, verify and debug the 9900 module in order to get a test simulation running properly. As the SBP 9989 constitutes a superset of instructions and signals of the TMS 9900, an SBP 9989 implementation in N.mPc was created by adding the appropriate instructions and signals to the 9900 module. Finally the SBP 9989 module was also checked in a test simulation.

1.3 Structure of the Report

Following this introduction, section 2 presents the salient features of TMS 9900 and the SBP 9989 architectures.

Section 3 gives some general information about the CAE tool N.mPc, making references to previous work and the existing N.mPc documentation. The main part of Section 3 outlines the debugging of the TMS 9900 hardware description provided in the N.mPc library, its upgrading to an SBP 9989 module and the addition of the new SBP 9989 instructions to the Metamicro assembler. Section 4 contains a discussion of the results obtained and conclusions reached concerning the 9900 and 9989. The appendices contain listings of an N.mPc test simulation and of various files needed to run it, as well as the protocol of a test simulation.

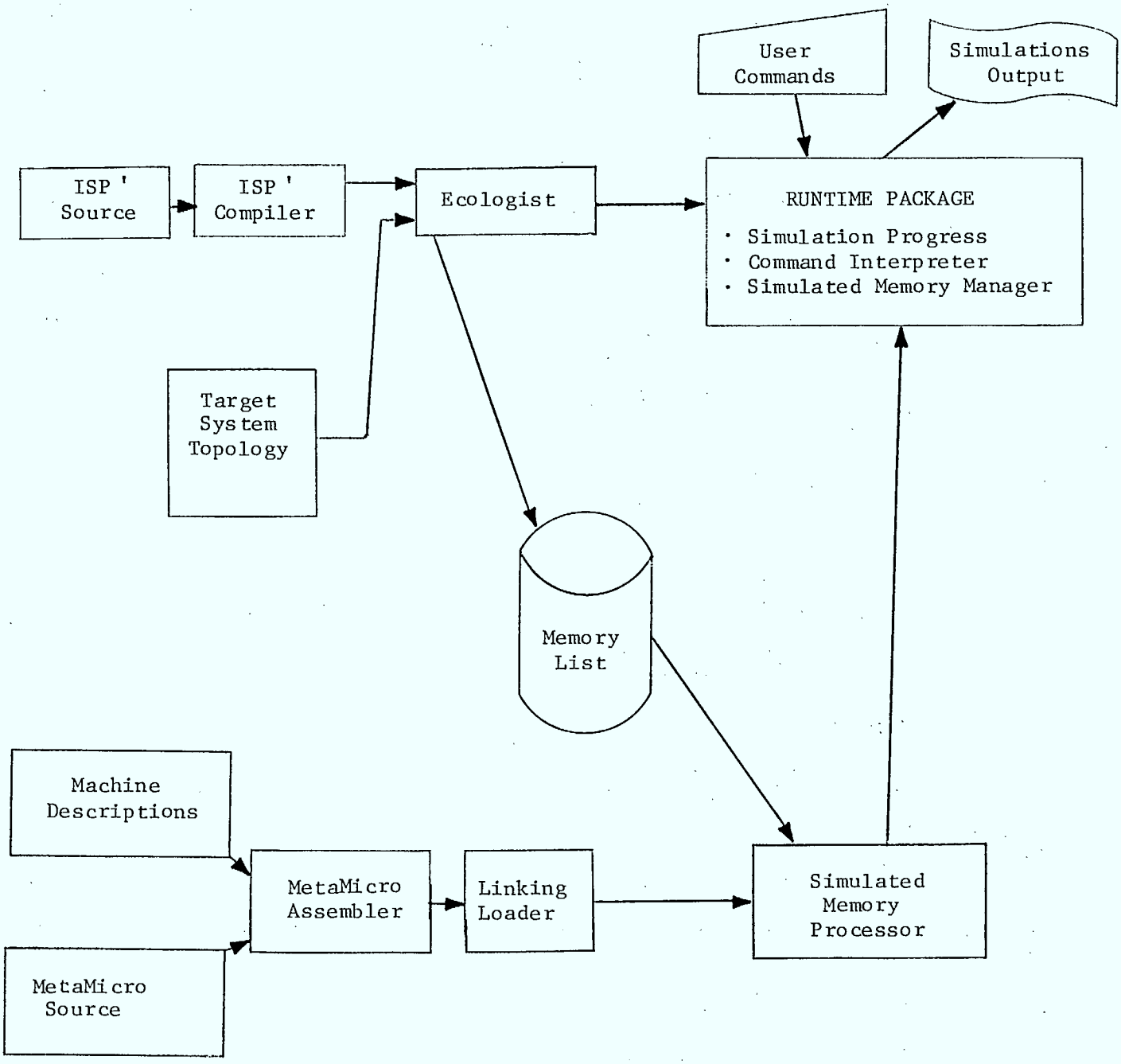


Figure 1.1 Main Components of the N.mPc System

2.0 TMS 9900 AND SBP 9989 MICROPROCESSORS

2.1 Architecture of the TMS 9900

The main features of the TMS 9900 are:

- 16 bit Instruction Word
- 64k Bytes of memory (on two pages to be swapped)
- 3.3 MHz speed
- Memory to Memory Architecture (Blocks of external memory designated as "work spaces" replace internal hardware registers)
- 16 general registers
- 16 prioritized interrupts
- DMA and programmed I/O facilities
- N-Channel Silicon-Gate Technology (NMOS)

The architecture of the 9900 relies on a 16 bit long memory word, thus consisting of two bytes which may be individually addressed by instructions. This situation is shown in Figure 2.1.

The 32K memory words (16 bits) are always addressed by even addresses while the 64K bytes may have odd or even addresses. The TMS 9900 and the SBP 9989 perform task switching very efficiently. This is due, in part, to their "memory-to-memory" architecture which dispenses with on-chip hardware registers and, instead, uses a contiguous block of 16 memory words. A workspace pointer is always pointing to the first of the 16 workspace registers which replace the processor internal hardware registers. A context switch, generated either by a program request or an interrupt, can be done quickly and easily since it is only a matter of exchanging the workspace pointer, the on-chip program counter and the on-chip status register. This feature of the 9900/9989 processors is depicted in Figure 2.2.

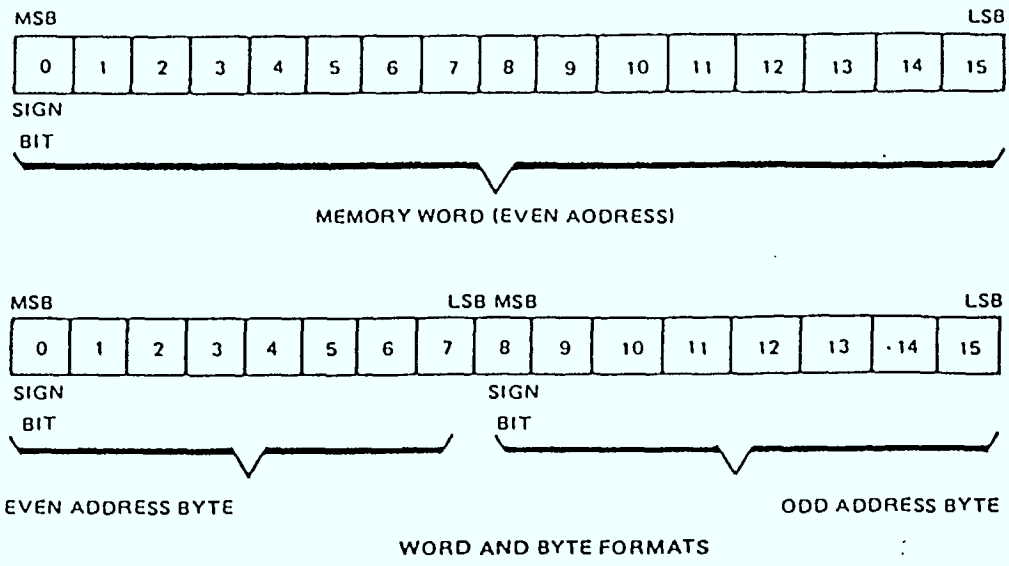
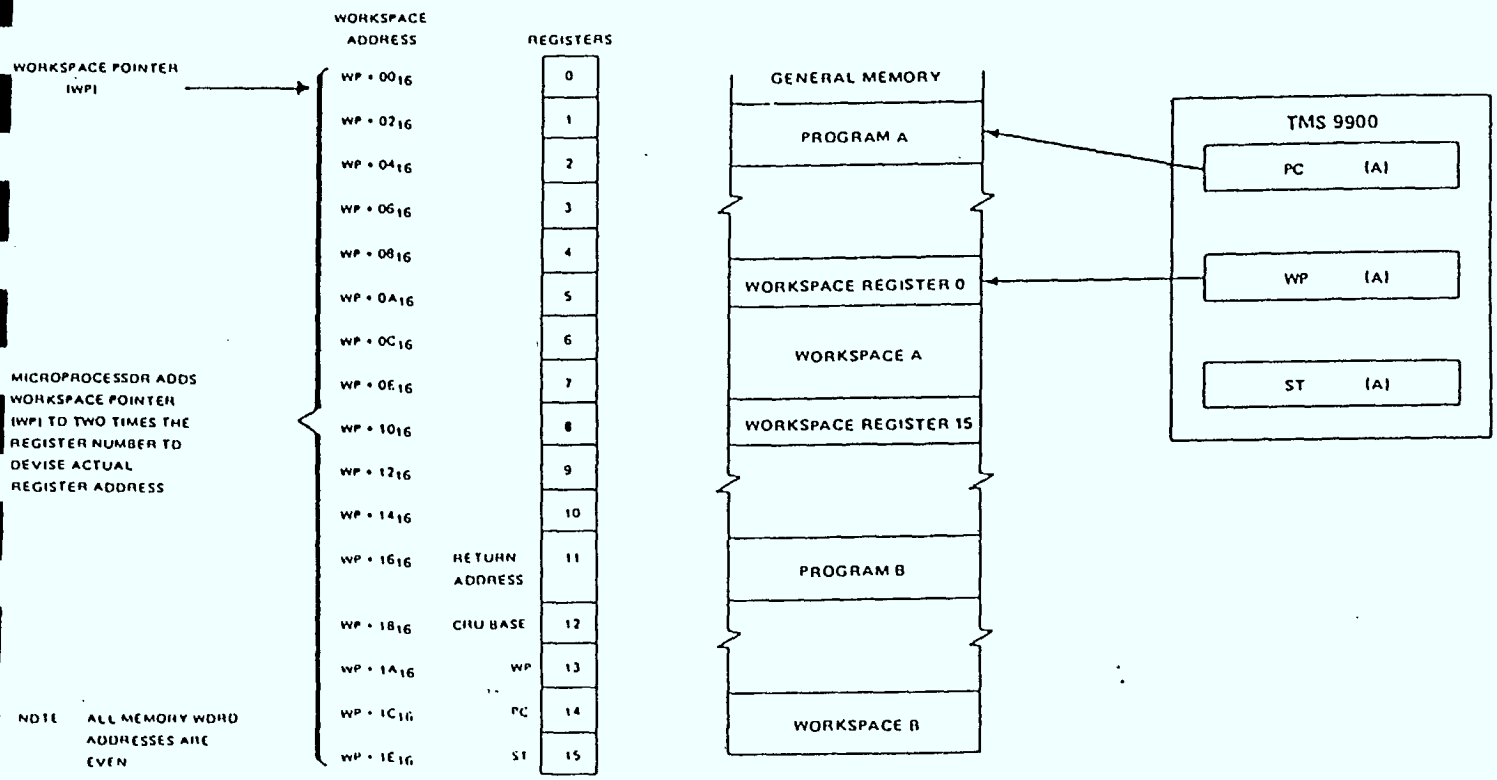


Figure 2.1 Memory Formats of the TMS 9900 (from TI Data Manual)



WORKSPACE POINTER AND REGISTERS

Figure 2.2 The TMS 9900 Workspace Concept (from TI Data Manual)

The first 64 words of the TMS 9900 memory space are reserved for interrupt and extended operation trap vectors. The rest of the 32K word general memory area is free for programs and "workspaces". The TMS 9900 can service up to sixteen different interrupts according to their priorities and supports direct memory access via extended operation instructions. A command driven, serial I/O unit (CRU) is also provided. Many of the TMS 9900's 69 instructions have the choice of five main addressing modes for their source and destination operands. Figure 2.3 shows the different ways the 9900's instructions access their operands.

2.2 Architecture of the SBP 9989

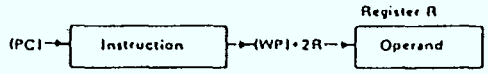
The SBP 9989 microprocessor is a military standard, upgraded version of the TI9900. The SBP 9989 has all the features of the 9900 plus four more instructions, 4 new signals, higher speed and is implemented in "integrated -injection circuit logic" technology. This "I^2L" technology gives the 9989 its high reliability, radiation hardness, temperature range and the low power consumption which make it suitable for spacecraft applications. The new features of the SBP 9989 as compared to the 9900 are listed below:

1) An instruction set that includes all the 69 instructions of the 9900 plus signed multiply, signed divide, load workspace pointer (LWP) and load stack pointer (LST). LWP and LST allow the 9989 to load a complete software context from an external source.

2) A new output signal on the SBP 9989 called "Memory Map Enable" (MPEN) can be used to double the effective address space. The SBP 9989 can therefore address 128 Kbytes of memory using MPEN in a bank switching mode.

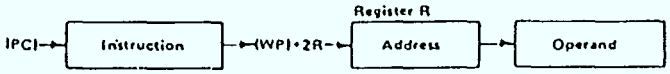
WORKSPACE REGISTER AOORESSING R

Workspace Register R contains the operand



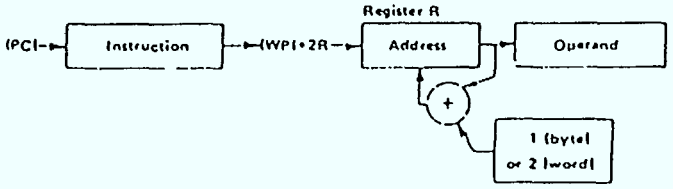
WORKSPACE REGISTER INDIRECT ADDRESSING *R

Workspace Register R contains the address of the operand.



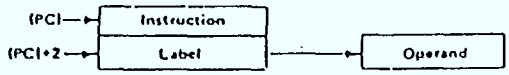
WORKSPACE REGISTER INDIRECT AUTO INCREMENT ADDRESSING *R+

Workspace Register R contains the address of the operand. After acquiring the operand, the contents of workspace register R are incremented.



SYMBOLIC (OIRECT) ADDRESSING @ LABEL

The word following the instruction contains the address of the operand.



INDEXED ADDRESSING @ TABLE (R)

The word following the instruction contains the base address. Workspace register R contains the index value. The sum of the base address and the index value results in the effective address of the operand.

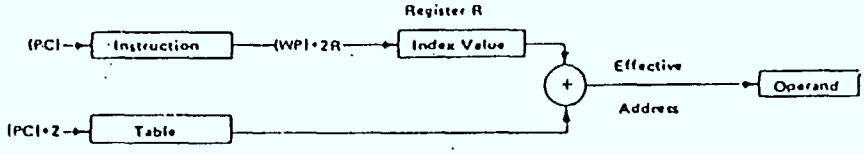


Figure 2.3 TMS 9900 Addressing Modes (from TI Data Manual)

This arrangement is not as convenient as a single linear space of the same size.

3) A "Multiprocessor Interlock" (MPILCK) allows coordination of the use of system resources by several processors.

4) "Interrupt Acknowledge" (INTACK) allows the 9989 to acknowledge an interrupt signal even at times when it does not have control of the system resources (bus, memory, etc.).

5) The "Extended Instruction Processor Present" (XIPP) Input Signal establishes a protocol for transfer of bus control between "host" and "slave" processors sharing the same memory.

6) The SBP 9989 may be driven by a 4.4 MHz clock (compared to 3.3 MHz of the 9900) and its instructions use 15-20% less microcycles than its "civilian" version. Texas Instruments claims that the throughput of the 9989 is twice that of the 9900. In view of the previous comments, that claim would appear quite optimistic.

The hardware block diagram in Figure 2.4 presents the MPEN, MPILCK, XIPP and INTACK signals. Memory architecture, addressing modes, I/O facilities, interrupt structure and memory map are identical in the TMS 9900 and SBP 9989 microprocessors. The SBP 9989 is a fully compatible version of the TMS 9900. It is faster, more reliable and better at number crunching and multiprocessing than the TMS 9900.

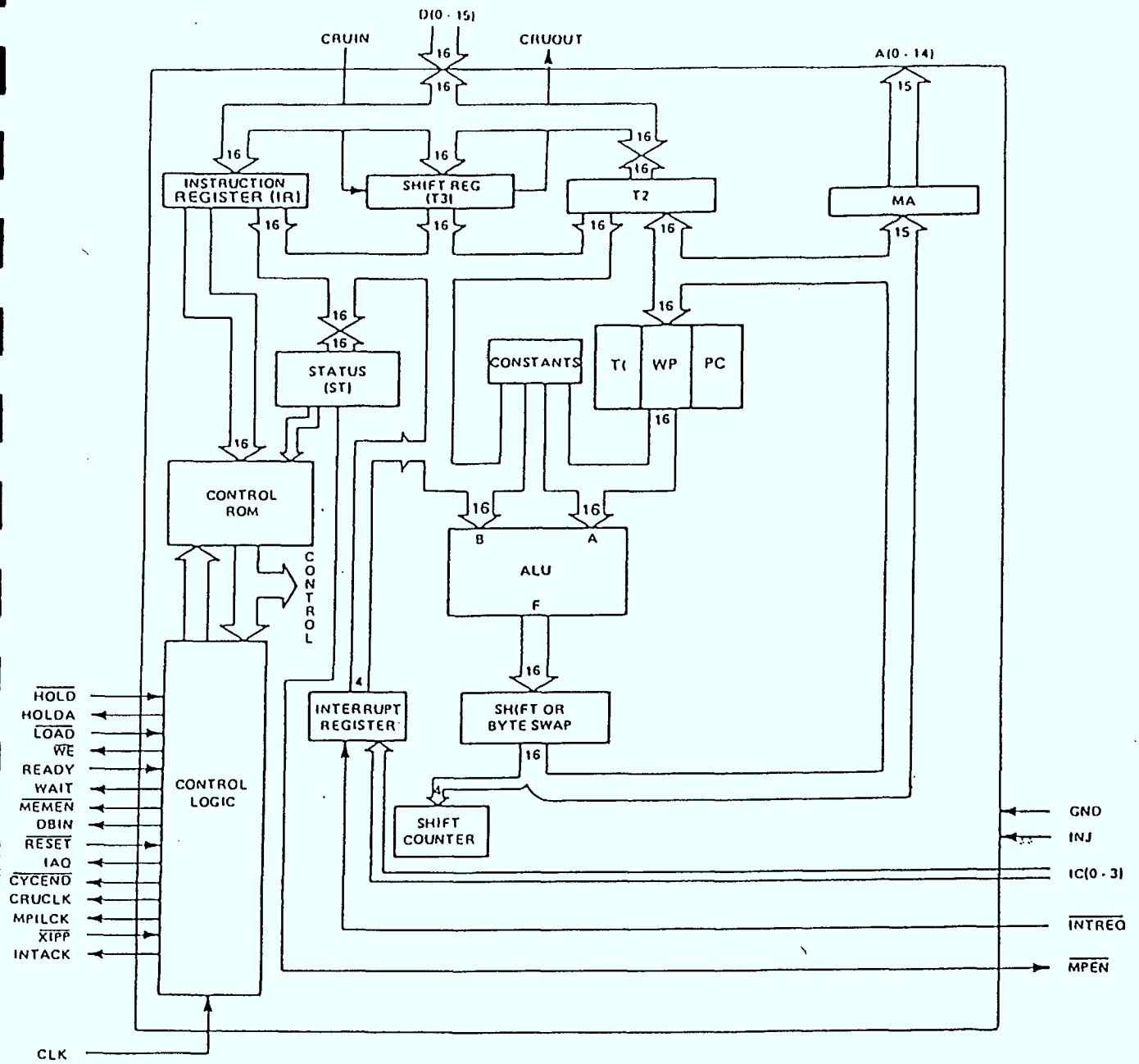


Figure 2.4 Hardware Block Diagram of the SBP 9989 (from TI Data Manual)

3.0 IMPLEMENTATION OF TMS 9900 AND SBP 9989 ON N.mPc

3.1 N.mPc File Naming Conventions

This section will be a brief reminder of the naming conventions in the N.mPc System. The files used in an N.mPc simulation are identified by a name which is chosen by the user and a compulsory suffix. Figure 3.1 shows the N.mPc System and the names of the files produced by the various elements of N.mPc. The following list describes the function of each file.

pgmname.m: A ".m" file is the source input file (= user program) to the metamicro assembler. A successful assembly produces a corresponding "pgmname.n" file.

pgmname.n: The intermediate file produced by the Metamicro assembler. Used by the Linking/Loader Allocator.

Lname.i: The source input to the Linking/Loader Interpreter. Contains the specification of the address resolution process for a given machine (stored in [.nmpc.softgen.llcf]).

Lname.a: The output of the Linking/Loader Interpreter. This file is used by the Allocator to direct the address resolution process.

1.out The output of the Allocator. Contains a real machine core image, suitable for simulation after processing by the "Simulated Memory Processor".

iname.isp The input of the ISP´ compiler, contains ISP´ source code describing a simulated piece of hardware.

iname.obj The output of the ISP´ compiler, corresponds to the iname.isp source input.

tname.t The topology file, the ecologist will build a program called "tname", which will be the executable simulation.

tname.s A symbol table file created by the ecologist, used by the runtime package. "tname" is the simulation name.

tname.f The memory list file, produced by the ecologist, and used by the simulated memory processor. Contains the names of all memories used in a simulation.

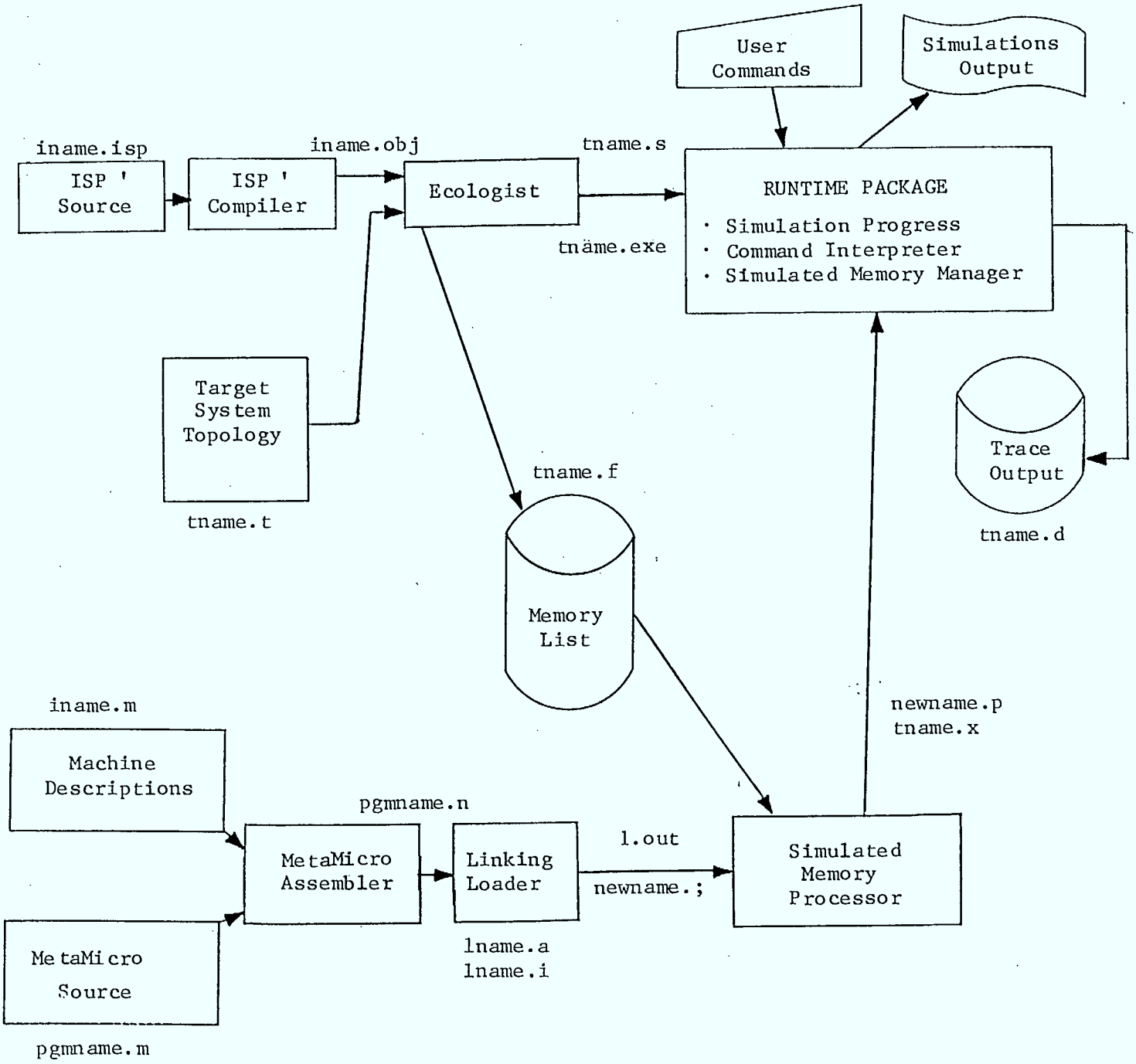


Figure 3.1 File Naming Conventions in the N.mPc System

newname.P — A processed file created by "smp". Corresponds to a previous Linking/Loader output file which has been renamed. Used by the simulation program and the Simulated Memory Editor.

newname.i — The contents of L.out are overwritten by each creation of a simulation. To share a "ready to run" set of files without going through the whole creation process the contents of 1.out have to be saved into another file (e.g. by doing Rename 1.out newname). The new file has to be declared in the topology file.

tname.x — Another smp output, containing global symbols from the Meta Micro; one per simulation.

tname.d — The simulation data file contains data to be processed by the post processor. This is a new feature of the VAX/VMS version of N.mPc.

tname.exe — The executable simulation program.

iname.m — The user programmed description of the (meta micro) assembler for the microprocessor being simulated; it has to be stored in the usual N.mPc directory structure (and can be reached by the path [.nmpc.softgen.mmpd]).

The N.mPc system seems quite complicated at first sight but all that is necessary to build a simulation is a hardware description (iname.isp), a topology file (tname.t), an assembler description (iname.m), the linking loader description (Lname.i) and some user program (pgnmname.m). These inputs are processed to form an executable simulation, which is controlled by "user commands". The N.mPc system has been used extensively by Intellitech, at first on PDP-11/UNIX and later on VAX/VMS. Interested readers are referred to the relevant reports [2,3], and the original N.mPc documentation [1, 8-16].

3.2 Implementation of the TMS 9900 on N.mPc

The success of an N.mPc simulation depends to a great extent on the quality of the descriptions of the various hardware modules required by the simulation. Ideally, one should be able to use existing hardware descriptions from an N.mPc library and concentrate on the simulation

building aspects.

The starting point for the implementation of the TMS 9900 on N.mPc was an ISP´ library description of that microprocessor provided with the N.mPc package. Several bugs, some of them of a subtle nature, existed in the ISP´ description and had to be corrected. The following paragraphs describe the nature of some of those bugs and the corrections that were subsequently made.

A test program designed to check the correct execution of the add instruction using all the five different addressing modes for source and destination operand showed a malfunction of the autoincrement indirect addressing mode. Instead of acquiring the source operand, the add procedure only took the address of the source operand in the addressing mode depicted in Figure 3.2.

The problem was clearly a logical bug in the corresponding procedure (op_5) mixing up the destination operand and its address in the case of auto increment addressing. Figure 3.3 shows the procedure "op 5", a part of the TMS 9900 description. The missing "read from memory" operation had to be added not only in the A (=add opcodes) instruction but also in the SZC (set zeroes corresponding; opcode 2) the S(=subtrack, opcode 3), the MOV (=move, opcode 6) and SOC (= set ones corresponding) instructions.

On the third line, only the address of the destination operand is read from a register (rreg) which was specified in the instruction. The reading of the operand from memory (using its address) is omitted. The improved "op 5" procedure now performs correct additions because (the changed) line 3 and (the new) line 4 read the destination operand from memory.

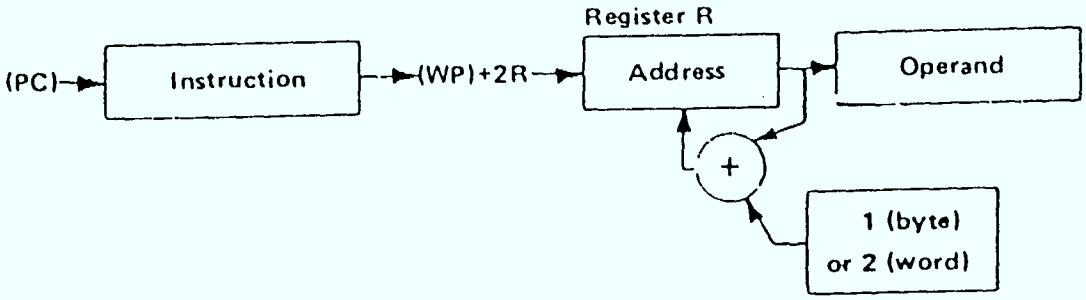


Figure 3.2 Workspace Register Indirect Auto-increment Addressing

```
/*******************************************************/
/*                                                     */
/* Executes opcode 5 group of instructions.            */
/* Instructions : A,AB src,dst                         */
/*                                                     */
/*******************************************************/

op_5 :=
  (
   S1 = src_opr(Sn,Smode,Size) .          !get first operand

   if Dmode eql 3 ! auto increment
     (
     D2 = 2*rreg(Dn,Size) . ;
     D1 = bus_read(D2,Size)
     )
   else
     D1 = src_opr(Dn,Dmode,Size) . ! Also dest.addr.,
   if Dmode eql 2                  ! so read last
     PC = PC - 2 .

   D2 = RC(D1,S1,add,Size) .              !compute the sum
   dst_opr(D2,Dn,Dmode,Size,se_f1) .      !store in dst
   0 = (S1<0> eql D1<0>) and (D2<0> neq D1<0>) .          !set

   delay(2) .                      !ALU cycles

   if Size eql B

      P = parity(D2<byte>)

  )
```

Figure 3.3 The Debugged "op 5" Procedure

The second major error in the TMS 9900 library description was the reading/writing to/from incorrect addresses in memory. In depth analysis of the problem showed that there was a difference in the address format generated by the "effective address calculation" procedure and the one assumed by the memory read/write procedures. Figure 3.4 shows the difference between these two address formats.

Once aware of this situation, corrective action was taken by changing the address recognition for source and destination operands. The stripping of the "least significant bit" of an addresss by the bus read/write procedures (assuming that the LSB is always zero) produced wrong addresses as the effective address provided was not in the assumed format.

To put the operand addresses in the appropriate format they are multiplied by two (equivalent to a "shift left" for binary numbers) prior to bus reading /writing of operands. To achieve this, only the address recognition in the "src-opr" and "dst-opr" procedures had to be changed by introducing a factor of two into the address recognition ("eff-addr" becomes "2* eff-addr"), as it is shown in Figure 3.5. The same problem as above was fixed in a similar fashion for the BWLP instruction. After the changes to the TMS 9900 library descriptions mentioned above, the test program documented in Appendix F began to execute properly.

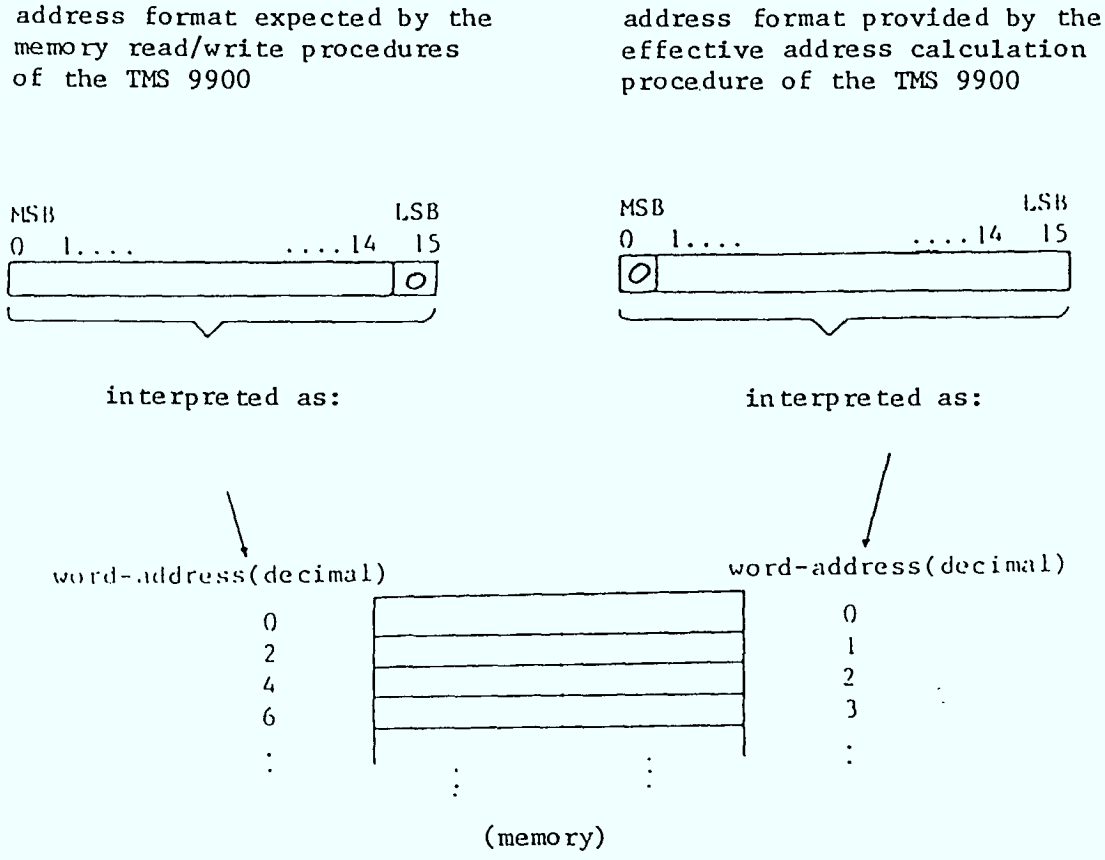


Figure 3.4 Different Address Formats (incorrectly) used in the TMS 9900 Library Description

```
/*****************************************************/
/*                                                   */
/* Fetches source operand.                           */
/* Input parameters: register number nibble,         */
/*                   addressing mode pair,           */
/*                   operand size bit.               */
/* Output parameters: source operand word.           */
/*                                                   */
/*****************************************************/

src_opr(rn<nib>,mode<pair>,size<bit>)<word> :=
  (
   case mode
     0      : ! register
      (
       src_opr = rreg(rn,size)
      )
     1,2,3 :/*
              indirect register,
              direct address,
              indexed,
              autoincrement
             */
      (
       A2 = 2*eff_addr(rn,mode,size).    !set the address
       src_opr = bus_read(A2,size)       !read from memory
      )
   esac
  )
```

Figure 3.5 The Debugged "src opr" Procedure

3.3 Implementation of the SBP 9989 on N.mPc

As the SBP 9989 is an upgraded TMS 9900 microprocessor, its implementation on N.mPc involved an upgrading of the TMS 9900 description mentioned in Section 3.2.

In programming design terms, four new instructions and four new signals ("ports") have to be added to the TMS 9900 description to make it an SBP 9989 module.

The added instructions are:

- signed multiply, performed by a procedure named "mpys"
- signed divide, performed by a procedure named "divs"
- load workspace pointer ("LWP") and load status register ("LST") enable the SBP 9989 to capture a complete software context from an external source

Three of the added system features ("ports") serve multiprocessing purposes:

- "multiprocessor interlock" (MPILCK), an output signal used to avoid access contention in resource sharing multiprocessing systems
- "interrupt acknowledge" (INTACK) enables the processor to acknowledge interrupts even while it is not in control of the system resources
- "Extended Instruction Processor Present" (XIPP) is an input signal needed for bus control transfer between the SBP 9989 and a co-processor.

The new output signal "MPEN" can either be manipulated to double the address space to 128 K bytes or be used (with a Memory Mapper) to allow access to 16 megabytes of memory. All these new signals and instructions are found in the listing of the SBP 9989 description in

Appendix A.1. Appendix A.2 contains the listing of the external memory description used with the SBP 9989. Appendices B, C, D contain listings of the topology, linking loader, and metamicro assembler description files needed to build an SBP 9989 simulation. Appendix E shows listings of the program (in different formats) used to test the SBP 9989 description. This test simulation is described in detail in Appendix F.

Following the conversion of the TMS 9900 ISP' description into a workable SBP 9989, testing activities began. Instructions from all opcode-groups were executed in short test programs, checking for correct contents of memory and/or registers before and after the execution of the instruction being tested. The initial situation was that a test program (test.m) could not be run because of bugs in the initial 9900 description. Upgrading of the TMS9900 description to an SBP9989 description along with the debugging mentioned above finally resulted in an SBP 9989 simulation on which the test.m program could be run successfully.

At this point it is important to stress that a working description of the SBP 9989 has been obtained. This description coupled with the appropriate meta micro assembler command files enables the N.mPc system to be used as a TMS 9900/SBP9989 development system. N.mPc is truly a programmable microprocessor development system.

4.0 DISCUSSION

The implementation of an SBP 9989 development system on N.mPc has been successfully accomplished and resulted in descriptions of the TMS 9900/SBP 9989 processors in the N.mPc hardware description language (ISP´), a programmable metamicro assembler for both processors and a test simulation.

Major portions of the SBP 9989 such as its decoding loop, its read/write procedures, and a number of its instructions were thoroughly checked, debugged and their functionality proven in actual test simulations. Due to lack of time however, not all the instructions of the SBP 9989 could be checked. In the event of future work with this processor description, further testing would have to be completed by checking the following components of the 9989 ISP´ description.

a) the "INTERRUPT" procedure;

b) the I/O procedures ("CRU_CLOCK,""CRU_TRANSFER");

c) the "LOAD", "RESET", "OVERFLOW TRAF", "SHIFT", "MPYS" and "DIVS" procedures;

d) most of the opcode 0 and opcode 1 instructions.

As in all N.mPc implementation activities, there is an extra benefit resulting from the careful study of the hardware to be implemented. In this case, the benefit is a thorough understanding of the workings of the SBP 9989 microprocessor. Based upon that understanding, the SBP 9989 can be compared with other microprocessors that are likely to be used in space applications. Examples of such processors include the Ferranti F100-L and the CMOS version of the Intel 8086.

The Ferranti F100-L is a 16 bit microprocessor implemented in bipolar logic for a high immunity to noise and radiation. This is an essential characteristic of any space qualified microprocessor. The F100-L does suffer, however, from a relative lack of advanced architectural features and powerful instructions. The memory map of the F100-L is also fairly non-standard and its handling of pointers can lead to confusion. The Ferranti F100-L was the subject of earlier work at Intellitech and the interested reader is referred to reference [2].

Recently, Harris announced a CMOS version of the Intel 8086 microprocessor. This new version would be suitable for space qualification and would then open the way for the use of the 8086 in spacecraft applications. The 8086 is a 16 bit microprocessor with a powerful instruction set and is compatible with Intel's family of peripheral device controllers. The 8086 is capable of a high throughput due, in part, to its speed and also to its pipelined bus unit/execution unit architecture. Other co-processors can be incorporated in an 8086 based system easily and they operate concurrently with the 8086. More information can be obtained from the relevant Intel literature and product description.

The SBP 9989 is also a 16 bit processor with a somewhat smaller address space than the 8086 (128 Kbytes vs 1Mbyte). In terms of instruction set, the 9989 fares better than the F100-L but does not offer the same versatility as the 8086. Furthermore, it does not feature a separate bus unit and execution unit architecture as the 8086. Speedwise, however, the 9989, although not as fast as the 8086, outperforms the Ferranti F100-L. This fact is substantiated by the recent switch of processors (i.e. from the F100-L to the 9989) in the British Aerospace L-Sat AOCS design. British Aerospace apparently found that the throughput of the F100-L was not sufficient to handle the load of a Spacecraft Microcomputer Module (SMM).

Having established the 9989 as a better alternative to the F100-L one may wish to compare the 9989 to the 8086. Apart from the fact that those two processors are of a 16 bit design, their respective architectures are quite different. Some of those differences are outlined in the four points below:

1. The SBP 9989 does not have on-chip general purpose registers. This slows down operations but facilitates task switching. Both the 9989 and the 8086 have flexible interrupt facilities reminiscent of those of the PDP-11. Interrupt service routines can be linked easily and priorities can be assigned with relative ease. (An 8086 would require a 8259 programmable interrupt controller). In the final analysis, the task switching advantage of the 9989 over the 8086 is offset by the latter's greater speed and by the fact that most interrupt service routines only save one or two on-chip registers in any case.

2. The Intel 8086 supports a segmented 1 Mbyte address space and a separate 64 Kbyte I/O space. This is vastly superior to the 9989's 128 Kbyte combined address and I/O space. To the 9989 defence, however, it must be added that in typical satellite applications, only a fraction of the 8086 address space would be used. Nevertheless, the large address space is there should modifications and enhancements be required at a later date.

3. Both processors provide mechanisms for the smooth integration of co-processors. An arithmetic co-processor is available for the SBP 9989; a wider choice of co-processors exists for the 8086 such as a numeric co-processor (8087), and an I/O co-processor (8089). Those co-processors are not currently space qualified but their usefulness in future spacecraft missions is easy to visualize.

4. The number and type of available support chips is much greater for the 8086 processor. This is due in part to the existing support chips for the 8 bit processors from Intel and to the greater popularity that the 8086 has enjoyed commercially.

While the SBP 9989 is a good processor in its own right, it appears that the coming availability of the 8086 in a CMOS version will make the 8086 the preferred choice in future implementations of on-board spacecraft control systems. In addition, software development utilities and packages are far more numerous for the 8086 than for the 9989. This software factor should be weighed very carefully when evaluating a given microprocessor.

It should be clear that due to rapid technology advances better processors are continually emerging. Therein lies the real advantage of using N.mPc in a design environment. Re-designs are easier and new processors can be incorporated and tested with greater ease than with traditional bread-boarding approaches. The development system for the SBP 9989 created using N.mPc is in keeping with that philosophy.

REFERENCES

[1] C.W. Rose, F.I. Parke, G.M. Ordy; "N.mPc: A Retrospective" and "The N.2 System" Case Western Reserve University, Cleveland, Ohio, June 1983.

[2] C. Laferriere, A. Lam; "N.mPc and its Utility for Spacecraft Applications" Report No. INT-83-47/1, Intellitech Canada Limited, January 1983.

[3] C. Laferriere, W.T. Brown, J.G. Ouimet, S.A. Mahmoud, "The Definition and Specification of an Integrated Set of CAE Tools for Spacecraft Multiprocessor System Design", Report No. INT-82-16, Intellitech Canada Limited, March 1982.

[4] Texas Instruments Semiconductor Products ; Master Selection Guide, 1982.

[5] TMS 9900 Microprocessor; Data Manual; August 1982.

[6] SBP 9989 Microprocessor, Data Manual, 1982.

[7] Texas Instruments; Software Development Handbook; 1981.

[8] G.M. Ordy, "N.mPc Release 2 Installation", Department of Computer Engineering, Case Western Reserve University, Cleveland, Ohio, November 1980.

[9] L.R. Rogers and G.M. Ordy, "N.mPc Metamicro User's Manual, version 3.1, Department of Computer engineering, Case Western Reserve University, Cleveland, Ohio, July 1980.

[10] C.W. Rose et al., "The N.mPc System Description Facility", Procedings of the 16th Design Automation conference (IEEE), pp. 520-528, June 1979.

[11] F.I. Parke et al., "The N.mPc Runtime Environment", Procedings of the 16th Design Automation Conference, (IEEE), pp. 529-536, June 1979.

[12] F.I. Parke , "An Introduction to the N.mPc Design Environment", Proceedings of the 16th Design Automation Conference (IEEE), pp. 513-519, June 1979.

[13] G.M. Ordy, "N.mPc Runtime User's Manual" Dep't of Computer Engineering, Case Western Reserve University, Cleveland, Ohio, Spring 1979.

[14] G.M. Ordy and F.I. Parke, "An Evaluation of the N.mPc Design System" Proceedings of the 16th Design Automation Conference (IEEE), pp. 537-541, June 1979.

[15] G.M. Ordy, "N.mPc Ecologist User´s Manual", Dep´t of Computer Engineering, Case Western Reserve University, Cleveland, Ohio, Spring 1978.

[16] R. Straubs, "N.mPc ISP´ User´s Manual", Dep´t of Computer Engineering, Case Western Reserve University, Cleveland, Ohio, 1978.

APPENDIX A: The Hardware Modules

A.1 The SBP9989 - Description (t9989.isp)

```
  pr t9989.isp

Dec 31 15:52 1983  t9989.isp Page 1

/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                           %
%          T E X A S   I N S T R U M E N T                                  %
%                                                                           %
%                S B P - 9 9 8 9                                            %
%                                                                           %
%                  Version 2.0                                              %
%                                                                           %
%   This ISP description is obtained by upgrading                           %
%   the description for the TMS 9900.  It includes                          %
%   all port definitions, external memory and bus                           %
%   interface, full DMA and interrupt capabilities.                         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   To make read/write-operations of words from/to                          %
%   memory function src_opr, dst_opr and the BLWP-                          %
%   instruction had to be changed(factor 2 in front                         %
%   of "eff_addr").Proper execution of the auto-in-                         %
%   crement addressing-mode was achieved by correc-                         %
%   ting op_2, op_3, op_5, op_6 and op_7.                                   %
%                                                                           %
%   Max Streit                                                              %
%   December 83                                                             %
%   Intellitech Canada Limited                                              %
%                                                                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

macro

/*********************************************************/
/*                                                       */
/* Symbolic names for register lengths.                  */
/*                                                       */
/*********************************************************/

  bit  =  0:0 &,
  pair =  0:1 &,
  nib  =  0:3 &,
  byte =  0:7 &,
  word = 0:15 &,
  lobyte = 8:15 &,
  addr = 0:14 &,

/*********************************************************/
/*                                                       */
/* Constants for size of operands.                       */
/*                                                       */
/*********************************************************/

  W = 0 &, ! Word
  B = 1 &, ! Byte


/*********************************************************/
/*                                                       */
/* Constants for addressing modes.                       */
/*                                                       */
```

```
/*****************************************************/

  r   = 0 &, ! register
  ir  = 1 &, ! indirect register
  da  = 2 &, ! direct address
  x   = 2 &, ! indexed

/*****************************************************/
/*                                                   */
/* Logic levels to select an option in a procedure.  */
/*                                                   */
/*****************************************************/

sub   = 0 &, ! operation was a subtraction
add   = 1 &, ! operation was an addition
de_fl = 0 &, ! deselect changing flags
se_fl = 1 &, ! select   changing flags

/*****************************************************/
/*                                                   */
/* Symbolic names for logic levels.                  */
/*                                                   */
/*****************************************************/

  false = 0 &,
  true  = 1 &,
  clear = 0 &,
  set   = 1 &,
  high = 1&,
  low = 0&,

/*****************************************************/
/*                                                   */
/*      Delay units for clock and timing.            */
/*                                                   */
/*****************************************************/

  clk = 1&,
  pulse = 2&,
  phase = 1&,

/*****************************************************/
/*                                                   */
/* Enforces sequential execution.                    */
/*                                                   */
/*****************************************************/

  . = ;next &;

state

/*****************************************************/
/*                                                   */
/* Declaration of registers.                         */
/*                                                   */
/*****************************************************/
```

```
  PC<word>,          ! Program Counter
  WP<word>,          ! Workspace Pointer
  ST<word>(0x0000),  ! STatus register

/*****************************************************/
/*                                                   */
/* Temporary registers.                              */
/*                                                   */
/*****************************************************/

  A1<word>, ! Address register 1
  A2<word>, ! Address register 2
  D1<word>, ! Destination register 1
  D2<word>, ! Destination register 2
  I1<word>, ! Instruction register 1
  I2<word>, ! Instruction register 2
  XR<word>, ! Extra Instr. register for X Remote instr.

  S1<word>, ! Source register 1
  mbr<word>,! Holds word last read from memory.
  in_vec<word>,! Holds addr. vect. for interrupt routin

  Load<bit>,  ! Flag set on load function
  Reset<bit>, ! Flag set on reset function
  Xipp<bit>,  ! Flag set on extended processor function
  Illop<bit>; ! Flag set on illegal opcode fetch

format

/*****************************************************/
/*                                                   */
/* Instruction register subfields.                   */
/*                                                   */
/*****************************************************/

  Opcode = I1<0:2>,   ! instruction set Opcode
  Size   = I1<3>,     ! operand Size
  Dmode  = I1<4:5>,   ! Destination mode
  Dn     = I1<6:9>,   ! Destination register number
  Smode  = I1<10:11>, ! Source mode
  Sn     = I1<12:15>, ! Source register number

/*****************************************************/
/*                                                   */
/* Status register subfields.                        */
/*                                                   */
/*****************************************************/

  LGT = ST<0>,      ! Logical Greater Than
  AGT = ST<1>,      ! Arithmatic Greater Than
  EQL = ST<2>,      ! EQuaL
  C   = ST<3>,      ! Carry
  O   = ST<4>,      ! Overflow
  P   = ST<5>,      ! Parity
  X   = ST<6>,      ! XOP
```

```
  IM  = ST<12:15>; ! Interrupt Mask


port

/*****************************************************/
/*                                                   */
/*             Chip pin description                  */
/*                                                   */
/*****************************************************/

     ! Data and address bus signals

     abus<0:14>,        ! address bus
     dbus<0:15>,        ! data bus

     ! Bus control signals

     dbin,              ! data bus in signal. high true
     memen(high),       ! memory enable. low true.
     we(high),               ! write enable. low true
     cruclk,            ! CRU clock signal from processor
     cruin,             ! CRU data in port for serial I/O
     cruout,            ! CRU data out port for serial I/O

     ! Memory control signals

     hold(high),        ! low true hold signal
     holda(low),        ! hold acknowledge. high true
     ready,             ! ready signal from memory
     pwait(low),        ! wait signal for memory operation
     mpen(high),        ! memory map enable output signal. low true

     ! Timing and control signals

     iaq,               ! instruction acquisition signal. high true
     load(high),             ! low true load signal to processor
     reset(high),            ! reset signal to processor. low true
     mpilck,            ! multiprocessor interlock output signal
     xipp(high),             ! extended instruction input signal. low true

     ! Interrupt control signals

     intreq(high),           ! low true interrupt request signal
     intcode<nib>,      ! interrupt priority code
     intack;            ! interrupt acknowledge output signal


/*****************************************************/
/*                                                   */
/*   Reads addressed word from memory                */
/*   Input parameters: 15 bit word address           */
/*   Output parameters: data word.                   */
/*                                                   */
/*****************************************************/
```

```
M_read(address<addr>)<word> :=
  (
    if hold eql low     !buses busy, ie. DMA active
      (
       holda = high;    !processor in hold state
       abus = clear; dbus = clear;      !make sure buses are clean
       while hold eql low delay(clk);   !wait until buses become available
       delay(clk);
       holda = low      !release from hold
      ) .
    abus = address<addr>;       !load the address bus
    dbin = high .               !data bus in and disable output
    memen = low;                !address bus valid
    delay(clk);
    if not ready                !memory not ready
      (
      delay(phase);
       pwait = high;            !processor in wait state because of a
       delay(3*phase);          !not ready condition from memory.
       while not ready delay(clk);      !wait until memory is ready
       delay(2*phase);
       pwait = low
      )
    else delay(phase) .
    M_read = dbus .             !read data bus
    wait(ready:trail)
    dbin = low .                !clear output signals
    abus = low .
    memen = high                !address bus idle
  )

/*****************************************************/
/*                                                   */
/*              Writes a word to memory              */
/*      Input parameters: 15 bit word address,       */
/*                        data word                  */
/*                                                   */
/*****************************************************/

M_write(data<word>,address<addr>) :=
    (
     if hold eql low    !bus not available
       (
        holda = high;   !processor in hold state
        abus = clear; dbus = clear;     !make sure buses are clean
        while hold eql low delay(clk);  !wait until buses available
        delay(clk);
        holda = low
       ) .
     dbus = data;       !load data bus
     abus = address;    !load address bus
     dbin = low .       !enable output
     memen = low;       !address bus valid
     delay(clk);
     we = low;          !write mode
```

```
      if not ready        !memory not ready
        (
         delay(phase);
         pwait = high;    !processor in wait state
         delay(3*phase);
         while not ready delay(clk);      !until memory becomes ready
         delay(2*phase);
         pwait = low
        )
      else delay(phase) .
      we = high;          !clean output signals
      wait(ready:trail)
      abus = low;
      dbus = low .
      memen = high        !address bus idle
     )

/*******************************************************/
/*                                                     */
/* Handles bus interface to read from memory.          */
/* Input parameters: address word,                     */
/*                   data size bit.                    */
/* Output parameters: data word.                       */
/*                                                     */
/*******************************************************/

bus_read(address<word>,size<bit>)<word> :=
  (

    mbr = M_read(address<addr>).        !last bit of address is striped
    case size
      W :       ! Word operation
        (
         bus_read = mbr<word>    !return the word
        )
      B :       ! Byte operation
        (
         case address<15>
           0 : ! even address
             (
              bus_read<byte> = mbr<byte>         !return <0:7> of word
             )
           1 : ! odd address
             (
              bus_read<byte> = mbr<lobyte>       !return <8:15> of word
             )
         esac
        )
    esac
  )

/*******************************************************/
/*                                                     */
/* Handles bus interface to write to memory.           */
/* Input parameters: data word,                        */
/*                   address word,                     */
```

```
/*                    data size bit.                          */
/*                                                            */
/************************************************************/

bus_write(data<word>,address<word>,size<bit>) :=
  (
    case size
      W : ! Word operation
        (
         M_write(data,address<addr>)     !write to memory. Last bit striped
        )
      B : ! Byte operation
        (
         case address<15>                !check the striped bit
           0 : ! even address
             (
              mbr<byte> = data<byte>.    !write only <0:7> of data word
              M_write(mbr,address<addr>)        !will overwrite <8:15> of word
             )
           1 : ! odd address
             (
              mbr<lobyte> = data<byte>. !write only <8:15> of data word
              M_write(mbr,address<addr>)        !will overwrite <0:7> of word
             )
         esac
        )
    esac
  )

/************************************************************/
/*                                                          */
/* Reads a register.                                        */
/* Input parameters: register number nibble,                */
/*                   data size bit.                         */
/* Output parameter: register value word.                   */
/*                                                          */
/************************************************************/

rreg(rn<nib>,size<bit>)<word> :=
  (
   state mar<word>;      ! temp reg. for address

   mar = WP + (2*(rn ext 8)) .   !get address of register

   mbr = M_read(mar<addr>) .     !read from memory and store temporarily

   case size
     W : ! Word
      (
       rreg = mbr
      )
     B : ! Byte
      (
       rreg<byte> = mbr<byte>    !return only bit <0:7>
      )
   esac
```

```
  )

/*****************************************************/
/*                                                   */
/* Writes a register.                                */
/* Input parameters: data word,                      */
/*                   register number nibble,         */
/*                   data size bit.                  */
/*                                                   */
/*****************************************************/

wreg(data<word>,rn<nib>,size<bit>) :=
  (
   state mar<word> ; ! temp. reg. for address

   mar = WP + (2*(rn ext 8)) .  !get address of reg in memory

   case size
     W : ! Word
      (
       M_write(data,mar<addr>)  !write into memory
      )
     B : ! Byte
      (
       mbr<byte> = data<byte> . !write byte into memory
       M_write(mbr,mar<addr>)
      )
   esac;
  )

/*****************************************************/
/*                                                   */
/* Calculates effective address.                     */
/* Effective address is derived depending on the     */
/* the addressing modes : indirect, direct, indexed  */
/* and autoincrement.                                */
/* Input parameters: register number nibble,         */
/*                   addressing mode pair,           */
/*                   data size bit.                  */
/* Output parameters: effective address word.        */
/*                                                   */
/*****************************************************/

eff_addr(rn<nib>,mode<pair>,size<bit>)<word> :=
  (
   delay(clk) .
   case mode            !addressing modes
     1 : ! indirect register
      (
       eff_addr = rreg(rn,W)    !rn contains the address
      )
     2 : ! direct address, indexed
      (
       A1 = bus_read(PC,W) .    !get address pointed by PC
       PC = PC + 2 .            !increment PC
       case rn          !indexed registers
```

```
          0       :       !zero means direct address
           (
            eff_addr = A1
           )
          default :       !any register can be used as index
           (
            eff_addr = rreg(rn,W) + A1 .
            delay(clk)
           )
        esac
       )
      3 : ! autoincrement
       (
        A1 = rreg(rn,W) .          !register contains the address
        eff_addr = A1 .            !return the address
        delay(clk) .

        A1 = size =>
               A1 + 1              !increment by one for byte operation
             else
               A1 + 2 .            !by two for word operation.
        wreg(A1,rn,W)              !restore the new value of register
       )
    esac
   )

/*****************************************************/
/*                                                   */
/* Fetches source operand.                           */
/* Input parameters: register number nibble,         */
/*                   addressing mode pair,           */
/*                   operand size bit.               */
/* Output parameters: source operand word.           */
/*                                                   */
/*****************************************************/

src_opr(rn<nib>,mode<pair>,size<bit>)<word> :=
   (
    case mode
      0     : ! register
       (
        src_opr = rreg(rn,size)
       )
      1,2,3 :/*
               indirect register,
               direct address,
               indexed,
               autoincrement
             */
       (
        A2 = 2*eff_addr(rn,mode,size).    !get the address
        src_opr = bus_read(A2,size)       !read from memory
       )
    esac
   )
```

```
/*****************************************************/
/*                                                   */
/* Changes LGT,AGT and EQL flags.                    */
/* Input parameters: data word,                      */
/*                   data size bit.                  */
/*                                                   */
/*****************************************************/

LAE(data<word>,size<bit>) :=
  (
   case size
     W : ! Word
      (
       LGT = data<word> neq 0 .          !ST<0>
       AGT = (data<0> eql 0) and (data<word> neq 0) .   !ST<1>
       EQL = data<word> eql 0 .          !ST<2>
      )
     B : ! Byte
      (
       LGT = data<byte> neq 0 .          !ST<0>
       AGT = (data<0> eql 0) and (data<byte> neq 0) .   !ST<1>
       EQL = data<byte> eql 0 .          !ST<2>
      )
   esac
  )

/*****************************************************/
/*                                                   */
/* Stores destination operand. Optionally changes    */
/* LGT,AGT and EQL flags.                            */
/* Input parameters: destination operand word,       */
/*                   register number nibble,         */
/*                   addressing mode pair,           */
/*                   operand size bit,               */
/*                   flag changing selection bit.    */
/*                                                   */
/*****************************************************/

dst_opr(data<word>,rn<nib>,mode<pair>,
        size<bit>,fl_chang<bit>) :=
  (
   if fl_chang eql se_fl
     (
      LAE(data,size)
     ) .
   case mode
     0     : ! register
      (
       wreg(data,rn,size) .
      )
     1,2,3 : /* indirect register,
                direct address,
                indexed,
                autoincrement
             */
      (
```

```
        A2 = 2*eff_addr(rn,mode,size) .
        bus_write(data,A2,size)
       )
    esac;
   )

/*******************************************************/
/*                                                     */
/* Fetches second instruction word.                    */
/*                                                     */
/*******************************************************/

fetch_2 :=
   (
    I2 = bus_read(PC,W).
    PC = PC + 2
   )

/*******************************************************/
/*                                                     */
/* Determines truth of a condition code.               */
/* Input parameters: condition code nibble.            */
/* Output parameters: condition code true/false bit.   */
/*                                                     */
/*******************************************************/

condi_code(rn<nib>)<bit> :=
   (
    condi_code = case rn
                   0  : true
                   1  : (not AGT) and (not EQL)
                   2  : (not LGT) or (EQL)
                   3  : EQL
                   4  : (LGT) or (EQL)
                   5  : AGT
                   6  : not EQL
                   7  : not C
                   8  : C
                   9  : not O
                   10 : (not LGT) and (not EQL)
                   11 : (LGT) and (not EQL)
                   12 : P
                 esac
   )

/*******************************************************/
/*                                                     */
/* Gets Result of an operation and changes C flag.     */
/* Handles two operation : add & sub.                  */
/* Input parameters: operand1 word,                    */
/*                   operand2 word,                    */
/*                   add/sub mode selection bit.       */
/* Output parameter: result word.                      */
/*                                                     */
/*******************************************************/
```

```
RC(op1<word>,op2<word>,operation<bit>,
                          size<bit>)<word> :=
  (
   state OP1<0:16>,
         OP2<0:16>,
         RESULT<0:16>;
   OP1<0:16> = op1<0:15> ext 17 .
   OP2<0:16> = op2<0:15> ext 17 .
   case operation
     sub : ! subtraction
      (
       case size
         B : ! Byte
          (
           RESULT<0:8> = OP1<0:8> - OP2<0:8>
          )
         W : ! Word
          (
           RESULT<0:16> = OP1<0:16> - OP2<0:16>
          )
       esac
      )
     add : ! addition
      (
       case size
         B : ! Byte
          (
           RESULT<0:8> = OP1<0:8> + OP2<0:8>
          )
         W : ! Word
          (
           RESULT<0:16> = OP1<0:16> + OP2<0:16>
          )
       esac
      )
   esac .
   RC = RESULT<1:16> .
   C = RESULT<0>
  )


/*******************************************************/
/*                                                     */
/*  Does a context switch by fetching new WP and PC    */
/*  while saving the present WP, PC, and status (ST)   */
/*  in the new workspace.                              */
/*  Input parameters: Address for context switch       */
/*                                                     */
/*******************************************************/


context_switch(address<word>) :=
               (
                delay(5) .                          !ALU cycles
                D1 = WP ; S1 = PC ; D2 = ST .       !store temporarily
                WP = bus_read(address,W) .          !assign new values
```

```
                PC = bus_read(address+2,W) .
                wreg(D1,13,W) .                    !store in memory
                wreg(S1,14,W) .
                wreg(D2,15,W) .
               )

/*****************************************************/
/*                                                   */
/*   Executes the context switch on a load signal    */
/*   by loading WP with vector FFFC and PC with      */
/*   FFFE.Also sets interrupt mask in ST to zero,    */
/*   clears status bit 7 to 11                       */
/*   and clears the load signal flag(Load).          */
/*                                                   */
/*****************************************************/

LOAD :=
     (
      delay(5) .                   !ALU cycles
      intack=set .                 !disable interrupt before context switch
      context_switch(0xfffc) .
      intack=low .                 !enable
      IM = 0 .                     !4 bit interrupt mask
      ST<7:11>= 0 .
      mpen = not ST<8>.
      Load = clear;                !reset load indicator
     )


/*****************************************************/
/*                                                   */
/*   Executes reset signal context switch using      */
/*   vectors 0000 for WP and 0002 for PC             */
/*   Interrupt mask in ST set to zero.The reset      */
/*   signal flag(Reset) is also cleared.             */
/*                                                   */
/*****************************************************/

RESET :=
      (
       delay(5) .                        !ALU cycles
       intack = set .              !disable interrupt
       context_switch(0x0000) .
       intack = low .              !enable
       ST=0 .
       mpen = not ST<8>.           !clear interrupt mask
       Reset = clear               !clear reset indicator
      )


/*****************************************************/
/*                                                   */
/*   Executes overflow trap context switch using     */
/*   vectors 0008 for WP and 000a.                   */
/*   Interrupt mask in ST set to one.                */
/*   ST<7:11> is also cleared.                       */
```

```
/*                                                              */
/****************************************************************/

OVERFLOW_TRAP :=
     (
        intack=set .              !enable interrupt
        context_switch(0x0008) .
        intack=low .              !disable interrupt
        IM=1 .                    !set interrupt mask to 1
        ST<7:11>=0 .
        mpen=not ST<8>
     )


/****************************************************************/
/*                                                              */
/*   This is the context switch at the start of                 */
/*   interrupt service by the processor.The service             */
/*   routine vectors are calculated using the value             */
/*   of the interrupt code at the intcode port.                 */
/*                                                              */
/****************************************************************/

INTERRUPT :=
          (
           in_vec = (4 * (intcode ext 16)) .
           intack=set .
           context_switch(in_vec) .
           intack = low .
           ST<7:11>=0 .
           mpen=not ST<8> .
           if intcode eql 0
             IM = 0
           else
             IM = intcode - 1     !set interrupt mask
          )

/****************************************************************/
/*                                                              */
/*    Provides clock pulses from processor for all              */
/*    required CRU operations and for user defined              */
/*    instructions like CKON,CKOFF,LREX etc                     */
/*    Clock pulse output is from cruclk port                    */
/*                                                              */
/****************************************************************/

CRU_CLOCK :=
    (
     delay(clk) .
     cruclk = high .
     delay(pulse) .
     cruclk = low .
     delay(clk-pulse)
    )
```

```
/*****************************************************/
/*                                                   */
/*    Transfers one bit of data from processor       */
/*    through cruout line for serial transfer        */
/*    of data. Input parameters: Register with,      */
/*                               bit count           */
/*                               shift start loc.    */
/*    Data assumed to be in register D1              */
/*                                                   */
/*****************************************************/

cru_transfer(bicnt_reg<nib>,start<nib>) :=
   (
    do
      (
       memen = high;
       if (start ext 8) eql 7    ! byte transfer
         (
          cruout = D1<7>
         )
       else    ! word transfer
         (
          cruout = D1<15>
         ) .
       CRU_CLOCK .

       bicnt_reg = bicnt_reg - 1;

       delay(clk)

       if bicnt_reg neq 0
         (
          D1 = D1 /: logical 1 ;
          abus = abus + 1
         )
      )
    until (bicnt_reg eql 0) .
    cruout = low
   )


/*****************************************************/
/*                                                   */
/*    Procedure to perform shift operations of       */
/*    processor.                                     */
/*    Input parameters: Data to be shifted must      */
/*                      be in register D1.           */
/*                      Register with shift cnt.     */
/*                      Shift function code.         */
/*                                                   */
/*****************************************************/


SHIFT(count_reg<nib>,functn<pair>) :=
          (
            case functn
```

```
                0 :     ! SRA src,count
                  (
                   do
                     (
                      C = D1<15> .
                      D1 = D1 /: arith 1 .
                      count_reg = count_reg - 1
                     )
                   until (count_reg eql 0) .
                   dst_opr(D1,Sn,r,W,se_f1)
                  )

                1 :     !  SRL src,count
                  (
                   do
                     (
                      C = D1<15> .
                      D1 = D1 /: logical 1 .
                      count_reg = count_reg - 1 .
                     )
                   until (count_reg eql 0) .
                   dst_opr(D1,Sn,r,W,se_f1)
                  )

                2 :     !  SLA src,count
                  (
                   do
                     (
                      C = D1<0> .
                      D1 = D1 *: arith 1 .
                      count_reg = count_reg - 1
                     )
                   until (count_reg eql 0) .
                   dst_opr(D1,Sn,r,W,se_f1)
                  )

                3 :     ! SRC src,count
                  (
                   if count_reg eql 0
                     D2 = D1 /: rotate 16
                   else
                     D2 = D1 /: rotate (count_reg ext 8) .
                   C = D1<0> .
                   dst_opr(D2,Sn,r,W,se_f1)
                  )
           esac .
           delay(3) .                    !ALU cycles
          )


/*************************************************/
/*                                               */
/* Signed multiplication of two 16 bit integers  */
/*      reg 0,1 = reg 0 * S1                     */
```

```
/*****************************************************/

mpys :=
  (
        state temp32<0:31>, temp16<0:15>;

        S1= src_opr(Sn, Smode, W) .
        temp16=rreg(0,W) .
        temp32 = S1 * temp16 .
        wreg(temp32<0:15>, 0 , W ) .            !reg 0 contains MSD
        wreg(temp32<16:31>, 1, W ) .            !reg 1 contains LSD

        if temp32 eql 0 ST<2>=set .
        if temp32 gtr 0 ST<1>=set .
        if temp32 neq 0 ST<0>=set .
        delay(23) .                     !ALU cycles
  )

/*****************************************************/
/*                                                   */
/* Signed division of two integers                   */
/*      Dividen is in reg 0,1 and is 32 bits         */
/*      Divisor is in any of the reg, 16 bits        */
/*      quotient is stored in reg 0                  */
/*      remainder is stored in reg 1                 */
/*****************************************************/

divs :=
  (
        state temp32<0:31>, temp16<0:15>;

        S1 = src_opr(Sn, Smode, W ) .   !get the divisor
        temp32<0:15> = rreg(0,W) .      !reg 0 contains MSD of dividend
        temp32<16:31> = rreg(1,W) .     !reg 1 contains LSD of dividend

        if (temp32<0> neq S1<0>) and ( abs (S1*32767) leq temp32)
                ST<4> = set .
        if S1 eql 0
          (
                ST<4> = set .
                delay(6) .                      !ALU cycles
          ) .

        if ST<4> eql 0
          (
              temp16=temp32/S1 .
              wreg(temp16,0,W) .        !Quotient
              if temp16 eql 0 ST<2>=set .
              if temp16 gtr 0 ST<1>=set .
              if temp16 neq 0 ST<0>=set .

              temp16 = temp32 mod S1 .
              wreg(temp16, 1, W) .      !remiander
              delay(23) .                       !ALU cycles
          )
  )
```

```
/*****************************************************/
/*                                                   */
/* Executes opcode 0 group of instructions.          */
/* Instructions : LI r,const          BLWP src       */
/*                AI r,const          B    src       */
/*                ANDI r,const        X    src       */
/*                ORI r,const         CLR  src       */
/*                CI r,const          NEG  src       */
/*                STWP r              INV  src       */
/*                STST r              INC  src       */
/*                LWPI const          INCT src       */
/*                LIMI const          DEC  src       */
/*                IDLE                DECT src       */
/*                RSET                BL   src       */
/*                RTWP                SWPB src       */
/*                CKON                SETO src       */
/*                CKOF                ABS  src       */
/*                LREX                TB   disp      */
/*                SBO  disp           SBZ  disp      */
/*                Static and dynamic shifts:         */
/*                 SRA  src,count  SLA  src,count    */
/*                 SRL  src,count  SRC  src,count    */
/*                All Jump instructions              */
/*                                                   */
/*****************************************************/

op_0 :=
  (
   state cnt<0:3>; ! counter for shift count

   case Size
     0 :
      (
       case Dmode
         0 :
          (
           case I1<7:10>
             0  : ! LI r,const
              (
               fetch_2 .
               dst_opr(I2,Sn,r,W,se_fl).
               delay(3) .                          !ALU cycles
           )
             1  : ! AI r,const
              (
               fetch_2 .
               D1 = src_opr(Sn,r,W) .
               D2 = RC(D1,I2,add,W) .
               0 = (D1<0> eql I2<0>) and
                     (D2<0> neq D1<0>);
               delay(3) .                          !ALU cycles
               dst_opr(D2,Sn,r,W,se_fl)
              )
             2  : ! ANDI r,const
              (
```

```
                fetch_2 .
                D1 = src_opr(Sn,r,W) .
                D2 = D1 and I2 .
                delay(3) .                              !ALU cycles
                dst_opr(D2,Sn,r,W,se_fl)
               )
              3  : ! ORI r,const
               (
                fetch_2 .
                D1 = src_opr(Sn,r,W) .
                D2 = D1 or I2 .
                delay(3) .                              !ALU cycles
                dst_opr(D2,Sn,r,W,se_fl)
               )
              4  : ! CI r,const   LST r     LWP r
               (
                 case I1<6>
                   0:  ! CI r,const
                     (
                        fetch_2 .
                        D1 = src_opr(Sn,r,W) .
                        D2 = D1 - I2 .
                        delay(3) .                              !ALU cycles
                        LGT = ((D1<0> eql 1) and (I2<0> eql 0)) or
                              ((D1<0> eql I2<0>) and (D2<0> eql 1)) .
                        AGT = ((D1<0> eql 0) and (I2<0> eql 1)) or
                              ((D1<0> eql I2<0>) and (D2<0> eql 1)) .
                        EQL = (D2 eql 0)
                     )
                   1 :  ! LST r and LWP r
                     (
                         case I1<11>
                           0 :  !  LST r
                             (
                                 ST = rreg(Sn,W ) .
                                 mpen=not ST<8> .
                                delay(3) .                              !ALU cycles
                     )
                                 1 :  !  LWP r
                                   (
                                      delay(3) .                        !ALU cycles
                                       WP = rreg(Sn,W)
                                   )
                              esac
                     )
                   esac
               )
              5  : ! STWP r
               (
                delay(2) .                              !ALU cycles
                dst_opr(WP,Sn,r,W,de_fl)
               )
              6  : ! STST r
               (
                delay(2) .                              !ALU cycles
                dst_opr(ST,Sn,r,W,de_fl)
```

```
        )
      7  : ! LWPI
        (
         fetch_2 .
         delay(4) .                          !ALU cycles
         WP = I2
        )
      8  : ! LIMI
        (
         fetch_2 .
         delay(4) .                          !ALU cycles
         IM = I2<12:15>
        )
      10 : ! IDLE
        (
         abus<0:2> = 2 .
         delay(3) .                          !ALU cycles
         CRU_CLOCK
        )
      11 : ! RSET
        (
         IM = 0;
         abus<0:2> = 3 .
         CRU_CLOCK
        )
      12 : ! RTWP  and  DIVS
        (
          case I1<6>
            0 :
              (
                  divs
              )
            1 : ! RTWP
              (
                 ST = src_opr(15,r,W) .
                 mpen=not ST<8> .
                 delay(4) .                          !ALU cycles
                 PC = src_opr(14,r,W) .
                 WP = src_opr(13,r,W)
              )
           esac
        )
      13 : ! CKON and DIVS
        (
          case I1<6>
            0 :
              (
                  divs
              )
            1 :
              (
                 abus<0:2> = 5 .
                 delay(3) .                          !ALU cycles
                 CRU_CLOCK
              )
          esac
```

```
            )
          14 : ! CKOF and MPYS
            (
              case I1<6>
                0 :
                  (
                      mpys
                  )
                1 :
                  (
                     abus<0:2> = 6 .
                     delay(3) .                          !ALU cycles
                     CRU_CLOCK
                  )
                esac
            )
          15 : ! LREX  and MPYS
            (
              case I1<6>
                0 :
                  (
                      mpys
                  )
                1 :
                  (
                     abus<0:2> = 7.
                     delay(3).                           !ALU cycles
                     CRU_CLOCK
                  )
                esac
            )
        esac
      )
    1 :
      (
       case Dn
         0  : ! BLWP src
          (
           A2 = 2 * eff_addr(Sn,Smode,W) .
           delay(6) .                          !ALU cycles
           context_switch(A2)
          )
         1  : ! B src
          (
           if Smode eql 0  ! register addressing
             PC = WP + (2 * (Sn ext 8))
           else
             PC = 2 * eff_addr(Sn,Smode,W) ;
           delay(3) .                          !ALU cycles
          )
         2  : ! X src
          (
           XR = src_opr(Sn,Smode,W) .
           I1 = XR ; XR = I1 .
           delay(1) .                          !ALU cycles
          )
```

```
            3  : ! CLR src
             (
              delay(2) .                          !ALU cycles
              dst_opr(0,Sn,Smode,W,de_fl)
             )
            4  : ! NEG src
             (
              D1 = src_opr(Sn,Smode,W) .
              D2 = RC(0,D1,sub,W) .
              dst_opr(D2,Sn,Smode,W,se_fl) .
              delay(3) .                          !ALU cycles
              0 = (D2 eql 0x8000)
             )
            5  : ! INV src
             (
              D2 = src_opr(Sn,Smode,W) .
              delay(2) .                          !ALU cycles
              dst_opr(not D2,Sn,Smode,W,se_fl)
             )
            6  : ! INC src
             (
              D1 = src_opr(Sn,Smode,W) .
              D2 = RC(D1,1,add,W) .
              dst_opr(D2,Sn,Smode,W,se_fl) .
              delay(2) .                          !ALU cycles
              0 = (D1<0> eql 0) and (D2<0> eql 1)
             )
            7  : ! INCT src
             (
              D1 = src_opr(Sn,Smode,W) .
              D2 = RC(D1,2,add,W) .
              dst_opr(D2,Sn,Smode,W,se_fl) .
              delay(2) .                          !ALU cycles
              0 = (D1<0> eql 0) and (D2<0> eql 1)
             )
            8  : ! DEC src
             (
              D1 = src_opr(Sn,Smode,W) .
              D2 = RC(D1,1,sub,W) .
              dst_opr(D2,Sn,Smode,W,se_fl) .
              delay(2) .                          !ALU cycles
              0 = (D1<0> eql 1) and (D2<0> eql 0)
             )
            9  : ! DECT src
             (
              D1 = src_opr(Sn,Smode,W) .
              D2 = RC(D2,2,sub,W) .
              dst_opr(D2,Sn,Smode,W,se_fl) .
              delay(2) .                          !ALU cycles
              0 = (D1<0> eql 1) and (D2<0> eql 0)
             )
            10 : ! BL src
             (
              dst_opr(PC,11,r,W,de_fl) .
              if Smode eql 0  ! register addressing
                PC = WP + (2 * (Sn ext 8))
```

```
            else
              PC = eff_addr(Sn,Smode,W) ;
            delay(3) .                          !ALU cycles
           )
          11 : ! SWPB src
           (
            D1 = src_opr(Sn,Smode,W) .
            D2<0:7> = D1<8:15> .
            D2<7:15> = D1<0:7> .
            delay(2) .                          !ALU cycles
            dst_opr(D2,Sn,Smode,W,de_f1)
           )
          12 : ! SETO src
           (
            delay(2) .                          !ALU cycles
            dst_opr(0xffff,Sn,Smode,W,de_f1)
           )
          13 : ! ABS src
           (
             mpilck= set .
            D1 = src_opr(Sn,Smode,W) .
            D2 = case D1<0>
                   0 : RC(0,D1,add,W)
                   1 : RC(0,D1,sub,W)
                 esac .
            dst_opr(D2,Sn,Smode,W,de_f1) .
            LGT = (D1 neq 0) .
            AGT = (D1<0> eql 0) and (D1 neq 0) .
            EQL = (D1 eql 0) .
             mpilck = low .
            0 = (D1 eql 0x8000);
            if D1<0> eql 1   delay(4)                !ALU cycles
            else delay(3) ;
           )
        esac
       )
      2 :
       (
        case I1<8:11>
          0       : ! dynamic shifts
           (
             S1 = rreg(0,W) .
             cnt = S1<12:15> .
             D1 = src_opr(Sn,r,W) .
             delay(3) .                         !ALU cycles

             SHIFT(cnt,I1<6:7>)
           )
          default : ! static shifts
           (
            cnt = I1<8:11> .
            D1 = src_opr(Sn,r,W) .

            SHIFT(cnt,I1<6:7>)
           )
        esac
```

```
          )
        esac
       )
      1 :
       (
        case I1<4:7>
          13       : ! SBO disp
           (
            D1 = rreg(12,W).
            D2<0:2> = 0;
            D2<3:14> = D1<3:14> + I1<8:15>.
            cruout = high; memen = high;
            abus = D2<addr>.
            CRU_CLOCK;
            abus = low;
            delay(3) .                      !ALU cycles
            cruout = low
           )
          14       : ! SBZ disp
           (
            D1 = rreg(12,W).
            D2<0:2> = 0;
            D2<3:14> = D1<3:14> + I1<8:15>.
            cruout = low; memen = high;
            abus = D2<addr>.
            CRU_CLOCK;
            abus = low;
            delay(3).                       !ALU cycles
           )
          15       : ! TB disp
           (
            D1 = rreg(12,W).
            D2<0:2> = 0;
            D2<3:14> = D1<3:14> + I1<8:15>.
            memen = high;
            abus = D2<addr>.
            delay(clk + (3*phase));
            ST<2> = cruin .
            delay(3) .                      !ALU cycles
            abus = low
           )
          default : ! Jcond disp
           (
            if condi_code(I1<4:7>)
               PC = PC + (2*(I1<8:15> sxt 16)) ;
            delay(3) .                      !ALU cycles
           )
        esac
       )
    esac
   )

/*****************************************************/
/*                                                   */
/* Executes opcode 1 group of instructions.          */
/* Instructions : COC src,r                          */
```

```
/*                  CZC src,r                                    */
/*                  XOR src,r                                    */
/*                  XOP src,r                                    */
/*                  LDCR src,count                               */
/*                  STCR src,count                               */
/*                  MPY src,r                                    */
/*                  DIV src,r                                    */
/*                                                               */
/*****************************************************************/

op_1 :=
  (
    state temp<0:15>,temp1<0:32>,temp2<0:16>,temp3<0:16>,
          cnt<0:3>;

    case (Size concat Dmode)
      0 : ! COC src,r
       (
        D1 = src_opr(Dn,r,W) .
        S1 = src_opr(Sn,r,W) .
        delay(3) .                              !ALU cycles
        EQL = ((D1 and S1) eql D1)
       )
      1 : ! CZC src,r
       (
        D1 = src_opr(Dn,r,W) .
        S1 = src_opr(Sn,r,W) .
        delay(3) .                              !ALU cycles
        EQL = ((D1 and S1) eql S1)
       )
      2 : ! XOR src,r
       (
        D1 = src_opr(Dn,r,W) .
        S1 = src_opr(Sn,r,W) .
        D2 = D1 xor S1 .
        delay(2) .                              !ALU cycles
        dst_opr(D2,Dn,r,W,se_f1)
       )
      3 : ! XOP src,r
       (
        ST<6> = high;
        temp = (4*Dn + 0x40) .
        context_switch(temp) .
        delay(7) .                              !ALU cycles
       wreg((eff_addr(Sn,Smode,W)),11,W)
       )
      4 : ! LDCR src,count
       (
        temp = rreg(12,W) . temp<0:2> = low .
        cnt = I1<6:9>.

        if ((cnt ext 8) leq 8) and ((cnt ext 8) gtr 0)
          (
           D2 = eff_addr(Sn,Smode,B).
           A1 = src_opr(Sn,Smode,B).
```

```
          if D2<15> eql 1
            (
             D1<lobyte> = A1<byte>.
             LAE(A1,B); P = parity(A1<byte>) .
             abus = temp<0:14> .
             cru_transfer(cnt,15)
            )
          else
            (
             D1<byte> = A1<byte>.
             LAE(A1,B); P = parity(A1<byte>).
             abus = temp<0:14> .
             cru_transfer(cnt,7)
            )
         )
        else
          (
           D2 = eff_addr(Sn,Smode,W).
           D1 = src_opr(Sn,Smode,W).
           LAE(D1,W).
           abus = temp<0:14> .
           cru_transfer(cnt,15)
          );
       delay(5) .                          !ALU cycles
       )
      5 : ! STCR src,count
       (
        temp = rreg(12,W) . temp<0:2> = clear .
        cnt = I1<6:9> .

        if ((cnt ext 8) leq 8) and ((cnt ext 8) gtr 0)
          (
           if Smode eql 3 ! auto increment
             D2 = rreg(Sn,Size)
           else
             D2 = src_opr(Sn,Smode,W). ! read destination
           if Smode eql 2              ! before storing
             PC = PC - 2 .

           abus = temp<0:14> .

           do
             (
              memen = high .
              delay(clk +(3*phase));
              D1<0> = cruin .
              delay(phase)

              cnt = cnt - 1 .

              delay(clk)
              if cnt neq 0
                (
                 D1 = D1 /: logical 1 ;
                 abus = abus + 1
                )
```

```
            )
           until (cnt eql 0).

           if (I1<6:9> ext 8) neq 8
             D1 = D1 /: logical (8 - (I1<6:9> ext 8)).
          dst_opr(D1,Sn,Smode,B,se_f1);
          P = parity(D1<byte>)
         )
       else
         (
          abus = temp<0:14> .

          do
            (
             memen = high .
             delay(clk+(3*phase));
             D1<0> = cruin .
             delay(phase)

             cnt = cnt - 1 .

             delay(clk)
             if cnt neq 0
               (
                D1 = D1 /: logical 1 ;
                abus = abus + 1
               )
            )
          until (cnt eql 0).
          if (I1<6:9> neq 0)
            D1 = D1 /: logical (16 - (I1<6:9> ext 8)) .
          dst_opr(D1,Sn,Smode,W,se_f1)
         );
        delay(8) .                          !ALU cycles
       )
     6 : ! MPY src,r
       (
         S1 = src_opr(Sn,Smode,W) .         !get first operand
         D1 = rreg(Dn,W) .                  !get second operand
         temp1<0:16> = 0; temp2<0> = 0; temp1<17:32> = S1;!clean scratch reg
         temp2<1:16> = D1 .

         temp1 = temp1 * temp2 . !compute product for unsigned multiplication

         if (Dn ext 8) eql 15    !special case if reg 15 is used
           (
            wreg(temp1<1:16>, 15, W) .            !most significant word
            M_write(temp1<17:32>, (WP+(2*16))) . !Least significant word
           )                                      !in memory next to reg
         else
           (
            wreg(temp1<1:16>, Dn, W) .            !most significant word
            wreg(temp1<17:32>, (Dn+1), W) .       !least significant word
           );
        delay(21) .                         !ALU cycles
       )
```

```
      7 : ! DIV src,r
       (
          S1 = src_opr(Sn,Smode,W) .       !get first operand
          temp1<0> = 0; temp2<0> = 0; temp3<0> = 0;       !clean scratch spaces
          temp2<1:16> = S1; temp1<1:16> = rreg(Dn,W) .    !copy & get operands
          temp3<1:16> = temp1<1:16>.

          if temp2 leq temp3       !divisor is leq than msd of dividend
            (
             delay(6) .                      !ALU cycles
             ST<4> = set             !set overflow flag
            )
          else
            (
             if (Dn ext 8) eql 15 !special case if reg 15 is used
               (
                temp1<17:32> = M_read(WP + 2*16) .  !get lsd of dividend

                temp3 = temp1 / temp2 .
                wreg(temp3<1:16>, 15, W) .        !store quotient in reg 15

                temp3 = temp1 mod temp2 .         !compute remainder
                M_write(temp3<1:16>, (WP + (2*16))) .     !store in next memory
               )

                temp3 = temp1 / temp2 .
                wreg(temp3<1:16>, Dn, W) .        !store quotient

                temp3 = temp1 mod temp2 .
                wreg(temp3<1:16>,Dn+1, W) .       !store remainder
              )
          );
         delay(23) .                         !ALU cycles
      )
    esac
)

/*****************************************************/
/*                                                   */
/* Executes opcode 2 group of instructions.          */
/* Instructions : SZC,SZCB src,dst                   */
/*                                                   */
/*****************************************************/

op_2 :=
    (
      S1 = src_opr(Sn,Smode,Size) .       !get first operand

    if Dmode eql 3     ! auto increment
      (
      D2 = 2*rreg(Dn,Size) .;
     D1 = bus_read(D2,Size)
      )
```

```
    else
      D1 = src_opr(Dn,Dmode,Size) . ! Also dest. address,
    if Dmode eql 2                  ! so read last
      PC = PC - 2 .

    D2 = D1 and (not S1) .          !compute set zero corresponding
    dst_opr(D2,Dn,Dmode,Size,se_fl) .    !store result D2 in dst
    delay(2) .                      !ALU cycles
    if Size eql B
       P = parity(D2<byte>)
  )

/*****************************************************/
/*                                                   */
/* Executes opcode 3 group of instructions.          */
/* Instructions : S,SB src,dst                       */
/*                                                   */
/*****************************************************/

op_3 :=
  (
    S1 = src_opr(Sn,Smode,Size) .        !get first operand

    if Dmode eql 3  ! auto increment
      (
      D2 = 2*rreg(Dn,Size) .;
      D1 = bus_read(D2,Size)
      )
    else
      D1 = src_opr(Dn,Dmode,Size) . ! Also dest.addr.,
    if Dmode eql 2                  ! so read last
      PC = PC - 2 .
    D2 = RC(D1,S1,sub,Size) .    !compute the difference

  dst_opr(D2,Dn,Dmode,Size,se_fl) .      !store in dst
    0 = (S1<0> neq D1<0>) and (D2<0> neq D1<0>) .        !set status bits
    delay(2) .                      !ALU cycles
    if Size eql B
       P = parity(D2<byte>)
  )

/*****************************************************/
/*                                                   */
/* Executes opcode 4 group of instructions.          */
/* Instructions : C,CB src,dst                       */
/*                                                   */
/*****************************************************/

op_4 :=
  (
    S1 = src_opr(Sn,Smode,Size) .        !get first operand
    D1 = src_opr(Dn,Dmode,Size) .        !get second operand
    D2 = D1 - S1 .                       !get the difference
    LGT = ((S1<0> eql 1) and (D1<0> eql 0)) or           !ST<0>
          ((S1<0> eql D1<0>) and (D2<0> eql 1)) .
    AGT = ((S1<0> eql 0) and (D1<0> eql 1)) or           !ST<1>
```

```
        ((S1<0> eql D1<0>) and (D2<0> eql 1)) .
   EQL = case Size                                          !ST<2>
           B : D2<byte> eql 0
           W : D2<word> eql 0
         esac .
   delay(3) .                       !ALU cycles
   if Size eql B
      P = parity(S1<byte>)
  )

/*******************************************************/
/*                                                     */
/* Executes opcode 5 group of instructions.            */
/* Instructions : A,AB src,dst                         */
/*                                                     */
/*******************************************************/

op_5 :=
  (
   S1 = src_opr(Sn,Smode,Size) .        !get first operand

   if Dmode eql 3 ! auto increment
     (
     D2 = 2*rreg(Dn,Size) . ;
    D1 = bus_read(D2,Size)
   )
   else
     D1 = src_opr(Dn,Dmode,Size) . ! Also dest.addr.,
   if Dmode eql 2                  ! so read last
     PC = PC - 2 .

   D2 = RC(D1,S1,add,Size) .            !compute the sum
   dst_opr(D2,Dn,Dmode,Size,se_f1) .    !store in dst
   0 = (S1<0> eql D1<0>) and (D2<0> neq D1<0>) .        !set status bit
   delay(2) .                       !ALU cycles
   if Size eql B
      P = parity(D2<byte>)
  )

/*******************************************************/
/*                                                     */
/* Executes opcode 6 group of instructions.            */
/* Instructions : MOV,MOVB src,dst                     */
/*                                                     */
/*******************************************************/

op_6 :=
  (
   D2 = src_opr(Sn,Smode,Size) .        !get the source operand

   if Dmode eql 3 ! auto increment
     (
     S1 = 2*rreg(Dn,Size) .;
     D1 = bus_read(S1,Size)
     )
   else
```

```
     D1 = src_opr(Dn, Dmode, Size) .! Read destination
    if Dmode eql 2                   ! before writing
      PC = PC - 2 .
   dst_opr(D2,Dn,Dmode,Size,se_f1) .  !store src in dst
   delay(2) .                      !ALU cycles
   if Size eql B
      P = parity(D2<byte>)
  )

/*******************************************************/
/*                                                     */
/* Executes opcode 7 group of instructions.            */
/* Instructions : SOC,SOCB src,dst                     */
/*                                                     */
/*******************************************************/

op_7 :=
  (
   S1 = src_opr(Sn,Smode,Size) .         !get first operand

   if Dmode eql 3 ! auto increment
     (
     D2 = 2*rreg(Dn,Size) .;
     D1 = bus_read(D2,Size)
     )
   else
     D1 = src_opr(Dn,Dmode,Size) . ! Also destination
   if Dmode eql 2                  ! address, so
     PC = PC - 2 .                 ! read last

   D2 = D1 or S1 .                 !compute set one corresponding
   dst_opr(D2,Dn,Dmode,Size,se_f1) .    !store result in dst
   delay(2) .                      !ALU cycles
   if Size eql B
      P = parity(D2<byte>)
  )


/*******************************************************/
/*                                                     */
/* Illegal Opcode check.                               */
/*                                                     */
/*******************************************************/

ILLOP_CHECK :=
    (
        Illop=0 .       !if illegal opcode is detected it will be set
        if (I1 geq 0x0000 and I1 leq 0x007f) Illop=set .
        if (I1 geq 0x00a0 and I1 leq 0x017f) Illop=set .
        if (I1 geq 0x0320 and I1 leq 0x033f) Illop=set .
        if (I1 geq 0x0780 and I1 leq 0x07ff) Illop=set .
        if (I1 geq 0x0c00 and I1 leq 0x0fff) Illop=set .
    )
/*******************************************************/
/*                                                     */
/* Fetches first instruction word.                     */
```

```
/*                                                          */
/************************************************************/

fetch_1 :=
  (
   iaq = high;           !indicate processor is fetching instruction
   I1 = bus_read(PC,W) .        !fetch instruction pointed by PC
   iaq =low;
   ILLOP_CHECK .         !check for illegal opcode
   PC = PC + 2           !increment PC
  )

/************************************************************/
/*                                                          */
/* Decodes instructions.                                    */
/*                                                          */
/************************************************************/

decode :=
  (
   delay(1) .                    !ALU cycles
   case Opcode
     0 : op_0
     1 : op_1
     2 : op_2
     3 : op_3
     4 : op_4
     5 : op_5
     6 : op_6
     7 : op_7
   esac
  )



/************************************************************/
/*                                                          */
/*  Decodes instruction fully and executes it.              */
/*                                                          */
/************************************************************/

execute :=
       (
        decode .
        if XR<0:9> eql 18 ! Execute remote instruction
          (
           decode .
           I1 = XR ; XR = I1 .
          )
       )
/************************************************************/
/*                                                          */
/* Execution of Extended Processor Instruction.             */
/*                                                          */
/************************************************************/
```

```
XIPACT :=
  (
   context_switch(0x0008) .      !reserve the environment
   delay(6) .                    !ALU cycles
   Illop = low .                 !reset illegal opcode indicator
   holda = high .                !processor is in hold state
   wait ( xipp : lead ) .        !wait until external processor is done
   holda = low .                 !exit from hold state
   WP = rreg ( 13,W ) .          !restore the environment
   PC = rreg ( 14,W ) .
   ST = rreg ( 15,W ) .
   mpen = not ST<8>
  )

/*****************************************************/
/*                                                   */
/* Main Processing Loop.                             */
/*                                                   */
/* 1- interrupt is recognized through intreq signal  */
/* 2- load,xipp and reset signals are processed by   */
/*    the when processors.                           */
/* 3- if illegal opcode is detected but xipp is not  */
/*    set, the process resumes normal operation.     */
/* 4- if overflow trap is enable and no interrupt    */
/*    request is pending, the overflow trap is       */
/*    executed.                                      */
/* 5- if idle instruction is executed, the processor */
/*    remains idle until either load, reset or       */
/*    interrupt event happens.                       */
/*                                                   */
/*****************************************************/

when(reset : trail) :=          !to recognize a reset signal
                    (
                     wait(reset : lead)
                     Reset = set
                    )

when(load : trail) :=           !to recognize a load signal
                   (
                    wait(load : lead)
                    Load = set
                   )

when(xipp :trail) :=            !to recognize extended instruction processor
                  (             !present signal
                   Xipp=set
                  )

main :=
    (
     fetch_1 .          !fetch instruction from memory
     if (Illop neq set and Xipp neq set)  execute       !decode and execute
     else
        (
           if Xipp XIPACT               !external processor takes over buses
```

```
            else
               (
                 delay(6) .                  !ALU cycles
                 if Illop                    !illegal opcode but no Xipp
                   (
                    Illop=low .              !reset illegal opcode indicator
                    context_switch(0x000a) .
                    I1<0:9>=0x010            !so to skip the checking below
                   )
               )
         ) ;
      if Load  LOAD
      else
         (
           if (I1<0:9> neq 0x010) and (I1<0:5> neq 0x0b) ! not BLWP or XOP
           (
             if (intreq eql low) and (intcode leq IM) INTERRUPT
             else
              (
                if (ST<4> eql set) and (ST<10> eql set) OVERFLOW_TRAP
                else
                (
                   if I1<0:10> eql 26   ! IDLE instruction, wait for an event
                   (
                      while ((load and reset) eql high) and (intreq eql high)
                        (
                           if (intcode ext 8) leq (IM ext 8) INTERRUPT
                           else CRU_CLOCK
                        ) .
                   ) ;
                   if Reset RESET .
                   if Load  LOAD
                )
              )
             );
         )
      )
```

APPENDIX A: The Hardware Modules

A.2 The Memory Module (timem.isp)

```
  cat timem.isp
/*************************************************************/
/*                                                           */
/*      MEMORY MODULE                                        */
/*                                                           */
/*      Designed to work with the Texas Instrument           */
/*      TMS 9989 processor.                                  */
/*      This memory Module does not support byte write       */
/*      operations.                                          */
/*                                                           */
/*      Albert Lam                                           */
/*      Intellitech Canada Limited        September 1983     */
/*                                                           */
/*************************************************************/

macro   HIGH    =       1&,
        LOW     =       0&,
        WORD    =       16&;

port    adbus<15>,      ! address bus
        dbus<WORD>,     ! data bus
        memen,          ! low true memory enable
        ready,          ! memory ready
        dbin,           ! data bus input signal
        we;             ! low true write enable

state   al<15>;         ! address latch

mem     me[0:255]<WORD>;        ! 16 bit wide memory

when ( memen:trail dbin eql HIGH) :=
        ( al = adbus ; next
          ready = HIGH ; next
          dbus = me[al]; next
          delay(3);
          ready = LOW ;
          dbus = 0;
        )

when ( memen:trail dbin eql LOW ) :=
        ( al = adbus ; next
          ready = HIGH ; next
          me[al] = dbus ; next
          delay(3);
          ready = LOW ;
        )

$
```

APPENDIX B: The Topology File (test.t)

```
%
/*****************************************************************/
/*                                                               */
/*        Topology file for initial test of TMS 9989 + memory        */
/*                                                               */
/*        Albert Lam                                             */
/*        Intellitech Canada Limited              09-SEP-83      */
/*                                                               */
/*****************************************************************/
%
signal
     adresb(15),       ! address bus
     datab(16),              ! data bus
     dbinb,               ! data bus in signal. high true
     memenb,               ! memory enable. low true.
     web,               ! write enable. low true
     cruclb,               ! CRU clock signal from processor
     cruinb,               ! CRU data in port for serial I/O
     cruoub,               ! CRU data out port for serial I/O
     holdb,              ! low true hold signal
     holdab,               ! hold acknowledge. high true
     readyb,               ! ready signal from memory
     pwaitb,               ! wait signal for memory operation
     mpenb,              ! memory map enable output signal. low true
     iaqb,              ! instruction acquisition signal. high true
     loadb,              ! low true load signal to processor
     resetb,               ! reset signal to processor. low true
     mpilcb,               ! multiprocessor interlock output signal
     xippb,              ! extended instruction input signal. low true
     intreb,               ! low true interrupt request signal
     intcob(4),               ! interrupt priority code
     intacb;               ! interrupt acknowledge output signal
%
/*****************************************************************/
/*                                                               */
/*        Processor descriptions                                    */
/*                                                               */
/*        pl = TMS 9989                                          */
/*        mem= memory module,initial max.core                    */
/*                                                               */
/*****************************************************************/
%
! T E X A S   I N S T R U M E N T   T M S - 9 9 8 9
processor        pl = "t9989.sim";
time delay        5 ns;
connections        abus = adresb,
                dbus = datab,
                dbin = dbinb,
                memen = memenb,
                we = web,
                cruclk = cruclb,
                cruin = cruinb,
                cruout = cruoub,
                hold = holdb,
                holda = holdab,
                ready = readyb,
                pwait = pwaitb,
                mpen = mpenb,
                iaq = iaqb,
                load = loadb,
                reset = resetb,
                mpilck = mpilcb,
                xipp = xippb,
                intreq = intreb,
                intcode = intcob,
                intack = intacb;
! M E M O R Y    M O D U L E
processor        mem = "timem.sim";
time delay        5ns;
connections        adbus = adresb,
                dbus = datab,
                memen = memenb,
                ready = readyb,
                dbin = dbinb,
                we = web;
initial                me = "max.core";
$
```

APPENDIX C: The Linking Loader Description (t9989.i)

```
        pr t9989.i

Sep 16 15:50 1983  t9989.i Page 1


!*****************************************************!
!*                                                   *!
!* t9900.i.                                          *!
!* Linking Loader description for Texas Instruments  *!
!* 9900 microprocessor.                              *!
!* No input requirements.                            *!
!* Generates t9900.a on compilation.                 *!
!* Use "inter" to compile.                           *!
!*                                                   *!
!* Author       : Samir S. Shah.                     *!
!* Date         : Summer 1980.                       *!
!* Modified     : Samir S. Shah.                     *!
!* Modification : The previous version addressed     *!
!*                memory by 16-bit words. It is      *!
!*                changed to byte addressing in      *!
!*                this version.                      *!
!* Date         : Sept 1980.                         *!
!*                                                   *!
!*****************************************************!

!*****************************************************!
!*                                                   *!
!* The memory is addressed in bytes. The default     *!
!* instruction length is two bytes. The maximum      *!
!* instruction length is six bytes.                  *!
!*                                                   *!
!*****************************************************!

instr
  I[3,1]<16>$

!*****************************************************!
!*                                                   *!
!* Alternative shorter names for instruction bytes.  *!
!*                                                   *!
!*****************************************************!

format
  I0 = I[0]<15:0>,
  I1 = I[1]<15:0>,
  I2 = I[2]<15:0>,

!*****************************************************!
!*                                                   *!
!* Subfields of instruction bytes.                   *!
!*                                                   *!
!*****************************************************!

  Opcode = I[0]<15:13>,
  BT     = I[0]<12>,
  TD     = I[0]<11:10>,
  DR0    = I[0]<9:8>,
  DR1    = I[0]<7:6>,
  TS     = I[0]<5:4>,
  SR     = I[0]<3:0>,
```

```
!*****************************************************!
!*                                                   *!
!* 8-bit extended opcode.                            *!
!*                                                   *!
!*****************************************************!

  extop = I[0]<7:0>,
  disp  = I[0]<7:0>,
  xtion = I[0]<7:5>,
  count = I[0]<7:4>$

!*****************************************************!
!*                                                   *!
!* Memory space declaration.                         *!
!*                                                   *!
!*****************************************************!

space
  <0:0x5fff>$


!*****************************************************!
!*                                                   *!
!* Unconditional Jump instruction for non-contiguous *!
!* memory allocation.                                *!
!*                                                   *!
!*****************************************************!

transfer
  {
   new
   Opcode = 0 $
   BT = 0 $
   TD = 1 $
   DR0 = 1 ^ -2 $
   DR1 = 1 $
   TS = 2 $
   SR = 0 $
   I1 = address $
   length = 2 $
  }

mode

!*****************************************************!
!*                                                   *!
!* Resolve one or both addresses.                    *!
!*                                                   *!
!*****************************************************!

case (length eql 3)      :
  if I1 eql 0 then { I1 = address[1] $ } $
  if I2 eql 0 then { I2 = address[2] $ } $
break $
esac,
case ((labelcnt eql 1) and
```

```
      (field eql I1))      :
  I1 = address $
break $
esac,
case ((labelcnt eql 1) and
      (field eql I2))      :
  I2 = address $
break $
esac,
case  (field eql disp)     :
  disp = (address + disp  - . - 1)  $
break $
esac,
default :
  I0 = address $
esac $
```

APPENDIX D: The "Meta Micro Assembler Description (t9989.m)"

```
  pr t9989.m

Sep 16 15:41 1983  t9989.m Page 1

!*****************************************************!
!*                                                   *!
!* t9900.m.                                          *!
!* metaMicro description file for Texas instruments  *!
!* 9900 microprocessor.                              *!
!* This file is included in the first line of        *!
!* assembler source code file. say source.m.         *!
!* It generates output file source.n if "micro" is   *!
!* used. If "mas" is used than it generates the      *!
!* following output files.                           *!
!* source.n : nodal output file.                     *!
!* source.l : assembler listing, logical addresses.  *!
!* source.L : assembler listing, both logical and    *!
!*            physical addresses and assembled       *!
!*            object code listings.                  *!
!* l.out    : assmbled object code core image.       *!
!* Use "micro" with "cater" and "merge" or use "mas".*!
!*                                                   *!
!* Author       : Samir S. Shah.                     *!
!* Date         : Summer 1980.                       *!
!* Modified     : Samir S. Shah.                     *!
!* Modification : The previous version addressed     *!
!*                memory in 16-bit words. It is      *!
!*                addressed in bytes in this version.*!
!* Date         : Sept 1980.                         *!
!* Modification : Converted to TMS9989. 4 more       *!
!*                instructions are added to reflect  *!
!*                this change.
!* Modified     : by Albert Lam, July 1983           *!
!*****************************************************!


!*****************************************************!
!*                                                   *!
!* The memory is addressed in bytes. The default     *!
!* instruction length is two bytes and the maximum   *!
!* instruction length is six bytes.                  *!
!*                                                   *!
!*****************************************************!

instr
  I[3,1]<16>$

!*****************************************************!
!*                                                   *!
!* Alternative shorter names for instruction bytes.  *!
!*                                                   *!
!*****************************************************!

format
  I0 = I[0]<15:0>,
  I1 = I[1]<15:0>,
  I2 = I[2]<15:0>,

!*****************************************************!
!*                                                   *!
```

```
!* Subfields of instruction bytes.                              *!
!*                                                              *!
!****************************************************************!

  Opcode = I[0]<15:13>,
  BT     = I[0]<12>,
  TD     = I[0]<11:10>,
  DR0    = I[0]<9:8>,
  DR1    = I[0]<7:6>,
  TS     = I[0]<5:4>,
  SR     = I[0]<3:0>,

!****************************************************************!
!*                                                              *!
!* 8-bit extended opcode.                                       *!
!*                                                              *!
!****************************************************************!

  extop = I[0]<15:8>,
  disp  = I[0]<7:0>,
  xtion = I[0]<7:5>,
  count = I[0]<7:4>$

macro

!****************************************************************!
!*                                                              *!
!* Register name macros.                                        *!
!*                                                              *!
!****************************************************************!

  R0  = 0  &,
  R1  = 1  &,
  R2  = 2  &,
  R3  = 3  &,
  R4  = 4  &,
  R5  = 5  &,
  R6  = 6  &,
  R7  = 7  &,
  R8  = 8  &,
  R9  = 9  &,
  R10 = 10 &,
  R11 = 11 &,
  R12 = 12 &,
  R13 = 13 &,
  R14 = 14 &,
  R15 = 15 &,

!****************************************************************!
!*                                                              *!
!* Addressing mode management.                                  *!
!* Abbreviations : RG = ReGister                                *!
!*                 RI = Register Indirect                       *!
!*                 SM = SyMbolic                                *!
!*                 IX = IndeXed                                 *!
!*                 AI = AutoIncrement                           *!
```

```
!*                                                              *!
!****************************************************************!

DRG (Rn) =
  TD = 0;
  DR0 = Rn ^ -2;
  DR1 = Rn &,

DRI (Rn) =
  TD = 1;
  DR0 = Rn ^ -2;
  DR1 = Rn &,

DSM (Ad) =
  TD = 2;
  DR0 = 0;
  DR1 = 0;
  if length neq 2 then {  I1 = Ad }
     else { I2 = Ad };
  length = length + 1 &,

DIX (X,Rn) =
  TD = 2;
  DR0 = Rn ^ -2;
  DR1 = Rn;
  if (length neq 2) then
    {
     I1 = X
    }
  else
    {
     I2 = X
    };
  length = length + 1 &,

DAI (Rn) =
  TD = 3;
  DR0 = Rn ^ -2;
  DR1 = Rn &,

SRG (Rn) =
  TS = 0;
  SR = Rn &,

SRI (Rn) =
  TS = 1;
  SR = Rn &,

SSM (Ad) =
  TS = 2;
  SR = 0;
  I1 = Ad;
  length = length + 1 &,

SIX (X,Rn) =
  TS = 2;
```

```
  SR = Rn;
  I1 = X;
  length = length + 1 &,

SAI (Rn) =
  TS = 3;
  SR = Rn &,

!*****************************************************!
!*                                                   *!
!* src and dst concatenation macros.                 *!
!*                                                   *!
!*****************************************************!

macS (src) =
  if 1 then {S}src &,

macD (dst) =
  if 1 then {D}dst &,

!*****************************************************!
!*                                                   *!
!* Format 1 instructions.                            *!
!*                                                   *!
!*****************************************************!

maclW (src,dst) =
  BT = 0;
  macS (src);
  macD (dst) &,

maclB (src,dst) =
  BT = 1;
  macS (src);
  macD (dst) &,

A (src,dst) =
  Opcode = 5;
  maclW (src,dst) $ &,

AB (src,dst) =
  Opcode = 5;
  maclB (src,dst) $ &,

C (src,dst) =
  Opcode = 4;
  maclW (src,dst) $ &,

CB (src,dst) =
  Opcode = 4;
  maclB (src,dst) $ &,

S (src,dst) =
  Opcode = 3;
  maclW (src,dst) $ &,
```

```
SB (src,dst) =
  Opcode = 3;
  macl8 (src,dst) $ &,

SOC (src,dst) =
  Opcode = 7;
  macl6 (src,dst) $ &,

SOCB (src,dst) =
  Opcode = 7;
  macl8 (src,dst) $ &,

SZC (src,dst) =
  Opcode = 2;
  macl6 (src,dst) $ &,

SZCB (src,dst) =
  Opcode = 2;
  macl8 (src,dst) $ &,

MOV (src,dst) =
  Opcode = 6;
  macl6 (src,dst) $ &,

MOVB (src,dst) =
  Opcode = 6;
  macl8 (src,dst) $ &,

!*************************************************!
!*                                               *!
!* Format 2 instructions.                        *!
!*                                               *!
!*************************************************!

mac2 (src,dstreg) =
  Opcode = 1;
  macS (src);
  DR0 = dstreg ^ -2;
  DR1 = dstreg &,

COC (src,dstreg) =
  BT = 0;
  TD = 0;
  mac2 (src,dstreg) $ &,

CZC (src,dstreg) =
  BT = 0;
  TD = 1;
  mac2 (src,dstreg) $ &,

XOR (src,dstreg) =
  BT = 0;
  TD = 2;
  mac2 (src,dstreg) $ &,

MPY (src,dstreg) =
```

```
  BT = 1;
  TD = 2;
  mac2 (src,dstreg) $ &,

DIV (src,dstreg) =
  BT = 1;
  TD = 3;
  mac2 (src,dstreg) $ &,


!*****************************************************!
!*                                                   *!
!* Format 3 instruction.                             *!
!*                                                   *!
!*****************************************************!

mac3 (src) =
        Opcode=0;
        BT=0;
        TD=0;
        macS(scr) &,

MPYS (src) =
        DR0 = 7 ^ -2;
        DR1 = 7;
        mac3(src) $ &,

DIVS (src) =
        DR0 = 6 ^ -2;
        DR1 = 6;
        mac3(src) $ &,


!*****************************************************!
!*                                                   *!
!* Format 4 instructions.                            *!
!*                                                   *!
!*****************************************************!

mac4 (src) =
  Opcode = 0;
  BT = 0;
  TD = 1;
  macS (src) &,

B (src) =
  DR0 = 1 ^ -2;
  DR1 = 1;
  mac4 (src) $ &,

BL (src) =
  DR0 = 10 ^ -2;
  DR1 = 10;
  mac4 (src) $ &,

BLWP (src) =
```

```
  DR0 = 0;
  DR1 = 0;
  mac4 (src) $ &,

CLR (src) =
  DR0 = 3 ^ -2;
  DR1 = 3;
  mac4 (src) $ &,

SETO (src) =
  DR0 = 12 ^ -2;
  DR1 = 12;
  mac4 (src) $ &,

INV (src) =
  DR0 = 5 ^ -2;
  DR1 = 5;
  mac4 (src) $ &,

NEG (src) =
  DR0 = 4 ^ -2;
  DR1 = 4;
  mac4 (src) $ &,

ABS (src) =
  DR0 = 13 ^ -2;
  DR1 = 13;
  mac4 (src) $ &,

SWPB (src) =
  DR0 = 11 ^ -2;
  DR1 = 11;
  mac4 (src) $ &,

INC (src) =
  DR0 = 6 ^ -2;
  DR1 = 6;
  mac4 (src) $ &,

INCT (src) =
  DR0 = 7 ^ -2;
  DR1 = 7;
  mac4 (src) $ &,

DEC (src) =
  DR0 = 8 ^ -2;
  DR1 = 8;
  mac4 (src) $ &,

DECT (src) =
  DR0 = 9 ^ -2;
  DR1 = 9;
  mac4 (src) $ &,

X (src) =
  DR0 = 2 ^ -2;
```

```
  DR1 = 2;
  mac4 (src) $ &,

!*****************************************************!
!*                                                   *!
!* Format 5 instructions.                            *!
!*                                                   *!
!*****************************************************!

mac5 (src,bits) =
  Opcode = 1;
  BT = 1;
  macS (src);
  DR0 = bits ^ -2;
  DR1 = bits &,

LDCR (src,bits) =
  TD = 0;
  mac5 (src,bits) $ &,

STCR (src,bits) =
  TD = 1;
  mac5 (src,bits) $ &,

!*****************************************************!
!*                                                   *!
!* Format 6 instructions.                            *!
!*                                                   *!
!*****************************************************!

SB0 (displ) =
  extop = 0x1d;
  disp = displ $ &,

SBZ (displ) =
  extop = 0x1e;
  disp = displ $ &,

TB (displ) =
  extop = 0x1f;
  disp = displ $ &,

!*****************************************************!
!*                                                   *!
!* Format 7 instructions.                            *!
!*                                                   *!
!*****************************************************!

JEQ (displ) =
  extop = 0x13;
  disp = displ $ &,

JGT (displ) =
  extop = 0x15;
  disp = displ $ &,
```

```
JH (displ) =
  extop = 0x1b;
  disp = displ $ &,

JHE (displ) =
  extop = 0x14;
  disp = displ $ &,

JL (displ) =
  extop = 0x1a;
  disp = displ $ &,

JLE (displ) =
  extop = 0x12;
  disp = displ $ &,

JLT (displ) =
  extop = 0x11;
  disp = displ $ &,

JMP (displ) =
  extop = 0x10;
  disp = displ $ &,

JNC (displ) =
  extop = 0x17;
  disp = displ $ &,

JNE (displ) =
  extop = 0x16;
  disp = displ $ &,

JNO (displ) =
  extop = 0x19;
  disp = displ $ &,

JOC (displ) =
  extop = 0x18;
  disp = displ $ &,

JOP (displ) =
  extop = 0x1c;
  disp = displ $ &,

!*************************************************!
!*                                               *!
!* Format 8 instructions.                        *!
!*                                               *!
!*************************************************!

SLA (srcreg,cnt) =
  extop = 0x0a;
  SR = srcreg;
  count = cnt $ &,

SRA (srcreg,cnt) =
```

```
  extop = 0x08;
  SR = srcreg;
  count = cnt $ &,

SRC (srcreg,cnt) =
  extop = 0x0b;
  SR = srcreg;
  count = cnt $ &,

SRL (srcreg,cnt) =
  extop = 0x09;
  SR = srcreg;
  count = cnt $ &,

!*****************************************************!
!*                                                   *!
!* Format 9 instructions.                            *!
!*                                                   *!
!*****************************************************!

mac9 (srcreg,operand) =
  extop = 0x02;
  SR = srcreg;
  I1 = operand;
  length = 2 &,

AI (srcreg,operand) =
  xtion = 1;
  mac9 (srcreg,operand) $ &,

ANDI (srcreg,operand) =
  xtion = 2;
  mac9 (srcreg,operand) $ &,

CI (srcreg,operand) =
  xtion = 4;
  mac9 (srcreg,operand) $ &,

LI (srcreg,operand) =
  xtion = 0;
  mac9 (srcreg,operand) $ &,

ORI (srcreg,operand) =
  xtion = 3;
  mac9 (srcreg,operand) $ &,

!*****************************************************!
!*                                                   *!
!* Format 10 instructions.                           *!
!*                                                   *!
!*****************************************************!

LWPI (operand) =
  extop = 0x02;
  xtion = 7;
  I1 = operand;
```

```
  length = 2 $ &,

LIMI (operand) =
  extop = 0x03;
  xtion = 0;
  I1 = operand;
  length = 2 $ &,



!*****************************************************!
!*                                                   *!
!* Format 11 instructions.                           *!
!*                                                   *!
!*****************************************************!

STST (srcreg) =
  extop = 0x02;
  xtion = 6;
  SR = srcreg $ &,

STWP (srcreg) =
  extop = 0x02;
  xtion = 5;
  SR = srcreg $ &,

LST (srcreg) =
  extop = 0;
  count=8;
  SR = srcreg $ &,

LWP (srcreg) =
  extop = 0;
  count = 9 ;
  SR = srcreg $ &,

!*****************************************************!
!*                                                   *!
!* Format 12 instruction.                            *!
!*                                                   *!
!*****************************************************!

RTWP =
  extop = 0x03;
  xtion = 4 $ &,


XOP (src,xopconst) =
  BT = 0;
  TD = 1;
  mac2 (src,xopconst) $ &,


!*****************************************************!
!*                                                   *!
!* Format 13 instructions.                           *!
```

```
!*                                                              *!
!****************************************************************!

macl3 =
  extop = 0x03 &,

IDLE =
  xtion = 2;
  macl3 $ &,

RSET =
  xtion = 3;
  macl3 $ &,

CKOF =
  xtion = 6;
  macl3 $ &,

CKON =
  xtion = 5;
  macl3 $ &,

LREX =
  xtion = 7;
  macl3 $ &,

!****************************************************************!
!*                                                              *!
!* Define Word psuedo-instruction.                              *!
!*                                                              *!
!****************************************************************!

DW (Word) =
  I0 = Word; $ &$
```

APPENDIX E: A Test Program (test.m; test.l, test.L)

```
  cat test.m
include /nmpc/softgen/mmpd/t9989.m $

begin

        .=0 $
        BLWP    (SM(128))
        .=128 $
getwp:  DW      (260)           !pointing to register area
getpc:  DW      (292)           !pointing to program area
        .=146 $                 !reserve spaces for registers
start:  MOV     (SM(n20),RG(R0))                !R0 is number of loop
        CLR     (RG(R1))                !reg # 1 is a counter
loop:   A       (SM(n3),SM(sum))                !add number to sum
        INC     (RG(R1))
        C       (RG(R1),RG(R0))         !compare number of loop with counter
        JNE     (loop)
        B       (SM(start))
sum:    DW      (0)
n20:    DW      (20)
n3:     DW      (3)
end
$
```

test.l:

```
1.      include /nmpc/softgen/mmpd/t9989.m $
2.
3.      begin
LEVEL: 1
4.      (0)
5.      (0)             .=0 $
6.      (0)             BLWP    (SM(128))
7.      (2)             .=128 $
8.      (2)     getwp:  DW      (260)           !pointing to register area
9.      (3)     getpc:  DW      (292)           !pointing to program area
10.     (4)             .=146 $                 !reserve spaces for registers
11.     (4)     start:  MOV     (SM(n20),RG(R0))                !R0 is number of loop
12.     (6)             CLR     (RG(R1))                !reg # 1 is a counter
13.     (7)     loop:   A       (SM(n3),SM(sum))                !add number to sum
14.     (10)            INC     (RG(R1))
15.     (11)            C       (RG(R1),RG(R0))         !compare number of loop with counter
16.     (12)            JNE     (loop)
17.     (13)            B       (SM(start))
18.     (15)    sum:    DW      (0)
19.     (16)    n20:    DW      (20)
20.     (17)    n3:     DW      (3)
21.     (18)    end
LEVEL: 1        Syntax Pass Completed.

Warnings:               0
Nonfatal errors:        0
Fatal errors:           0
$
```

test.L:

```
1.                                              include /nmpc/softgen/mmpd/t9989.m $
2.
3.                                              begin
LEVEL: 1
4.      (0)
5.      (0)                                             .=0 $
6.      (0)     (     0) C002040 000200       ]         BLWP    (SM(128))
7.      (2)                                             .=128 $
8.      (2)     (   200) C000404                ] getwp:  DW      (260)           !pointing to register area
9.      (3)     (   201) C000444                ] getpc:  DW      (292)           !pointing to program area
10.     (4)                                             .=146 $                 !reserve spaces for registers
11.     (4)     (   222) C140040 000236         ] start:  MOV     (SM(n20),RG(R0))                !R0 is number of loop
12.     (6)     (   224) C002301                ]         CLR     (RG(R1))                !reg # 1 is a counter
13.     (7)     (   225) C124040 000237 000235] loop:     A       (SM(n3),SM(sum))                !add number to sum
14.     (10)    (   230) C002601                ]         INC     (RG(R1))
15.     (11)    (   231) C100001                ]         C       (RG(R1),RG(R0))         !compare number of loop with counter
16.     (12)    (   232) C013372                ]         JNE     (loop)
17.     (13)    (   233) C002140 000222         ]         B       (SM(start))
18.     (15)    (   235) C000000                ] sum:    DW      (0)
19.     (16)    (   236) C000024                ] n20:    DW      (20)
20.     (17)    (   237) C000003                ] n3:     DW      (3)
21.     (18)                                            end
LEVEL: 1        Syntax Pass Completed.

Warnings:               0
Nonfatal errors:        0
Fatal errors:           0
$
```

APPENDIX F: A Test Simulation

This part of the appendix presents a test simulation for the SBP 9989 implementation on N.mPc. To recall the configuration of the elements see Figure 3.1 (Chapter 3.1). The "test.m" program first establishes a "workspace" starting at memory location 128, initializes the registers R0, R1 and increases R1, "sum" till R0 equals R1. The program works in an endless loop. The function of the test program is illustrated by the flowchart in Figure F.1. The linking of the test program is shown in Figure F.2.

R0, R1 are the first two of the sixteen workspace registers of the SBP 9989; "loop", "start", "sum" are labels.

The initialization procedure performs a context switch to spare the memory locations before 128 for later use by interrupt and extended operation trap vectors. Therefore the actual program is stored in memory locations 128 through 159.

The following shows how a test program is run on the SBP 9989, thereby checking the correct execution of all instructions used in the program.

The simulation is controlled by setting a breakpoint everytime a new instruction is loaded into the instruction register. By looking at the contents of the memory locations in question, we can check the correct execution of instructions.

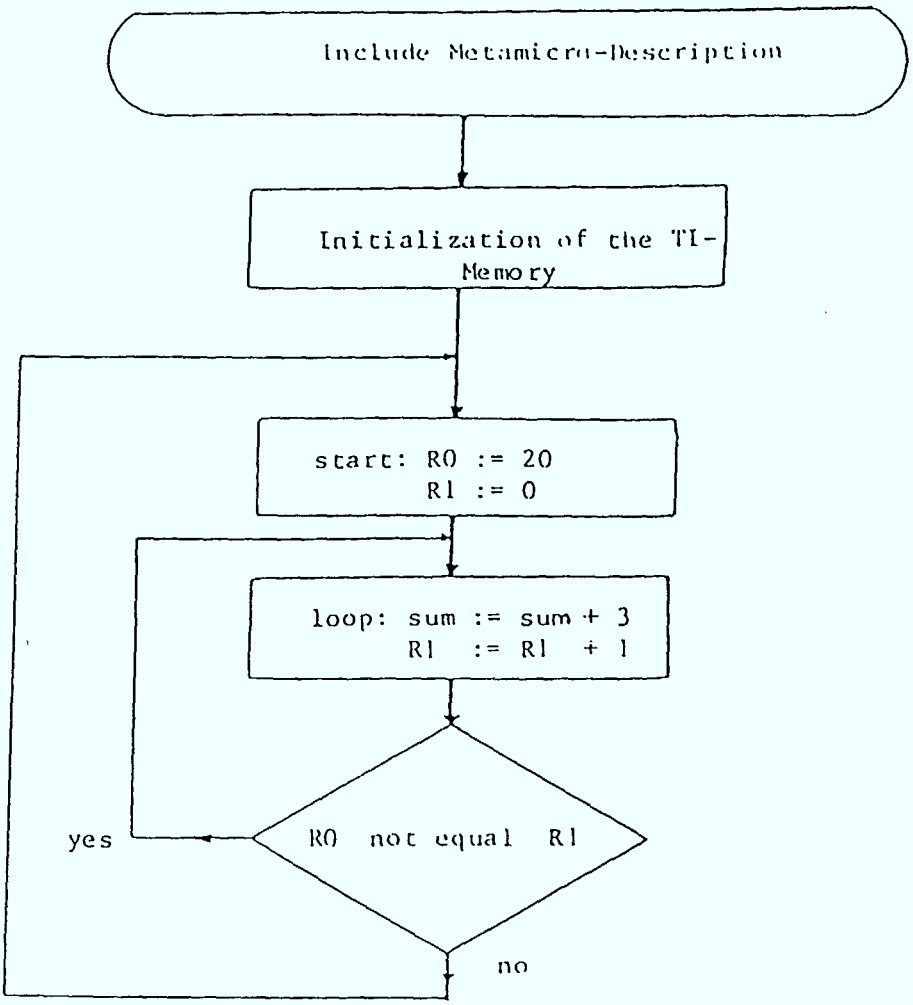


Figure F.1 Flowchart of the Test Program

```
  cat test.m
include /nmpc/softgen/mmpd/t9989.m $

begin

        .=0 $
        BLWP    (SM(128))
        .=128 $
setwp:  DW      (260)           !pointing to register area
setpc:  DW      (292)           !pointing to program area
        .=146 $                 !reserve spaces for registers
start:  MOV     (SM(n20),RG(R0))                !R0 is number of loop
        CLR     (RG(R1))                !reg # 1 is a counter
loop:   A       (SM(n3),SM(sum))                !add number to sum
        INC     (RG(R1))
        C       (RG(R1),RG(R0))         !compare number of loop with counter
        JNE     (loop)
        B       (SM(start))
sum:    DW      (0)
n20:    DW      (20)
n3:     DW      (3)
end
$
```

Figure F.2 Listing of the Test Program

```
  pwd
/nmpc/usr/t9989                    ! the working directory
$
$ ls                               ! files contained in the working-
max.core                           ! directory
max.core.p
t9989.isp
t9989.sim
test
test.L
test.d
test.f
test.l
test.m
test.n
test.s
test.t
test.x
timem.isp
timem.sim
$
$
$ test                             ! puts the simulation in the
N.mPc: test                        ! runtime-mode(different prompt-sign)
#
# memory   mem:me 0   4            !  examines contents of memory-locations
                  (0):     1056
                  (1):     128     ! in given limits; (0), (1) contain
                  (2):     0       ! the initialization(context-switch)
                  (3):     0
                  (4):     0
# memory mem:me   130 160          ! (130) to (145) contain the workspace-
                  (130):   0       ! registers(R0 - R15)
                  (131):   0
                  (132):   0
                  (133):   0
                  (134):   0
                  (135):   0
                  (136):   0
                  (137):   0
                  (138):   0
                  (139):   0
                  (140):   0
                  (141):   0
                  (142):   0
                  (143):   0
                  (144):   0
                  (145):   0
                  (146):   49184   ! (146) to (159) contain the user-
                  (147):   158     ! program
                  (148):   1217
                  (149):   43040
                  (150):   159
                  (151):   157
                  (152):   1409
                  (153):   32769
                  (154):   5882
                  (155):   1120
                  (156):   146
                  (157):   0
                  (158):   20
                  (159):   3
```

```
#
#
# states                                  ! shows the state(= value contained in)
                PC      0                 ! of all the temporary registers
                WP      0
                ST      0
                A1      0
                A2      0
                D1      0
                D2      0
                I1      0
                I2      0
                XR      0
                S1      0
                mbr     0
                in_vec  0
                Load    0
                Reset   0
                Xipp    0
                Illop   0
#
#
# ports                                   ! shows current values of all the
                abus    0                 ! communicating-elements(busses, con-
                dbus    0                 ! trol lines)
                dbin    0
                memen   0
                we      0
                cruclk  0
                cruin   0
                cruout  0
                hold    0
                holda   0
                ready   0
                mpwait  0
                mpen    0
                iaq     0
                load    0
                reset   0
                mpilck  0
                xipp    0
                intreq  0
                intcode 0
                intack  0
#
#
# repeat bkpt  :I1 change               ! stops the simulation after every
breakpoint number 1                     ! change of the instruction-register
# display  :I1 change                   ! displays the instruction-register
monitor number 2                        ! after every change without stopping
#                                       ! the simulation
```

```
#
# run                                          ! starts simulated execution of the
t: 15                                          ! program
        2 display  :I1 change = 1056           ! initial context-switch(BLWP) to
simulation halted by bkpt 1                    ! memory-location 128
(repeat bkpt  :I1 change)
#
#
# run
t: 185
        2 display  :I1 change = -16352         ! R0 := 20(MOV...)  is the next instruc-
simulation halted by bkpt 1                    ! tion to be executed
(repeat bkpt  :I1 change)
# states
                PC      292                    ! the context-switch has been executed
                WP      260
                ST      0
                A1      128
                A2      256
                D1      0
                D2      0
                I1      49184                  ! shows a different number than display
                I2      0                      ! because display interprets (binary)
                XR      0                      ! numbers as signed numbers while
                S1      4                      ! all other examination-commands treat
                mbr     49184                  ! numbers as absolute values
                in_vec  0
                Load    0
                Reset   0
                Xipp    0
                Illop   0
#
#
# memory mem:me 130 131
                (130):  0                      ! initial contents of R0, R1
                (131):  0
# deposit 22 mem:me[131]
#  memory mem:me 131
                (131):  22                     ! contents of R1 changed to a non-zero
#                                              ! value to check correct function of
#                                              ! "clear R0" in the next instruction
# run
t: 280
        2 display  :I1 change = 1217           ! "CLEAR R0" is the next instruction to be
simulation halted by bkpt 1                    ! executed
(repeat bkpt  :I1 change)
# memory mem:me 130
                (130):  20                     ! 20 was correctly moved into R0
#
#
# run
t: 325
        2 display  :I1 change = -22496         ! sum := sum + 3  is the next instruction
simulation halted by bkpt 1                    !  in assembly-language: A(SM(n3),SM(sum))
(repeat bkpt  :I1 change)
# memory  mem:me 131
                (131):  0                      ! R1 has correctly been clared
# mem mem:me 157
                (157):  0                      ! contents of "sum" before addition(of 3)
```

```
# run
t: 460
        2 display  :I1 change = 1409   ! "INCREMENT R1 is the next instruc-
simulation halted by bkpt 1            ! tion
(repeat bkpt  :I1 change)
# mem mem:me 157
                (157):  3              ! 3 has correctly been added to "sum"
#
# mem mem:me 131
                (131):  0              ! verification of the contents of R1
#                                      ! before incrementation
# run
t: 520
        2 display  :I1 change = -32767 ! "COMPARE R1,R0"  is the next
simulation halted by bkpt 1            ! instruction
(repeat bkpt  :I1 change)
#
# mem mem:me 131
                (131):  1              ! R1 has correctly been incremented by
#                                      ! one
# run
t: 585
        2 display  :I1 change = 5882   ! "jump to loop if R1,R0 not equal"
simulation halted by bkpt 1            ! is the next
(repeat bkpt  :I1 change)
# run
t: 620
        2 display  :I1 change = -22496 ! the program-counter jumps back to
simulation halted by bkpt 1            ! "loop" as R1=1 is not equal to R0=20.
(repeat bkpt  :I1 change)              ! the program therefore will go through
# run                                  ! the loop twenty times and then branch
t: 755                                 ! to "start", resetting R1 to 0
        2 display  :I1 change = 1409   !"increment R1" is the next instruction
simulation halted by bkpt 1
(repeat bkpt  :I1 change)
#
# mem mem :me  157
                (157):  6              ! 3 correctly added to "sum" in the se-
# run                                  ! cond passage through the loop
t: 815
        2 display  :I1 change = -32767 ! "compare R1,R0  is the next instruc-
simulation halted by bkpt 1            ! tion
(repeat bkpt  :I1 change)
# mem mem:me 131
                (131):  2              ! R1 correctly incremented
#
```

```
#
#
# whatis all
1 repeat bkpt   :I1 change
2 display   :I1 change
# rem 1
# rem 2
#
# bkpt   :I1 eql 49184
breakpoint number 3
#
# run
simulation halted by bkpt 3
(bkpt   :I1 eql 49184)
# mem mem:me 130 131
                  (130):   20
                  (131):   20
# mem mem:me 157
                  (157):   60
# repeat bkpt   :I1 change
breakpoint number 4
# run
simulation halted by bkpt 4
(repeat bkpt   :I1 change)
# mem mem:me 130 131
                  (130):   20
                  (131):   20
# run
simulation halted by bkpt 4
(repeat bkpt   :I1 change)
# mem mem:me   130 131
                  (130):   20
                  (131):   0
#
#
#
# quit
$
$
$
```

! shows all previously defined break-
! points and monitors

! removal of the present breakpoints and
! and monitors

! next simulation-stop only when it gets
! to "start"(= MOV = 49184) which means
! after passing the loop twenty times

! R1 = 20 means that the loop has been
! passed twenty times before jumping
! to "start"; 3 has been added to "sum"
! twenty times as well, so the 60
! in "sum"(= memory(157)) are correct

! the "clear R1"-instruction has been
! executed correctly , resetting the
! "loop-counter"R1 to 0 after 20 loops

! the simulation has been exited(the
! $-prompt is used again)

intellitech
Intellitech Canada Ltd
352 MacLaren Street
Ottawa, Ontario
K2P 0M6
(613)235-5126