1985

A design of a multi-process simulator for a new personal computer: a step in the right direction

Christopher N. Tisseverasinghe
University of Wollongong

A Design of a Multi-Process Simulator for a new Personal Computer

A Step in the Right Direction

A thesis submitted in partial fulfilment of the requirements for the award of the degree of

HONOURS MASTER OF SCIENCE
(Computing Science)

from

THE UNIVERSITY OF WOLLONGONG

by

CHRISTOPHER N. TISSEVERASINGHE BMath (ComSci) W'gong.

Department of Computing Science

1985

I hereby declare that I am the sole author of this thesis. I authorise the University of Wollongong to lend this thesis to other institutions or individuals for the purpose of scholarly reasearch.

ABSTRACT

This thesis presents the design and results of a simulator for a new multi-processor personal computer. A brief overview of the machine architecture will be presented as background.

The system design exploits the concept of multiple processes to support separation of concerns and facilitate modularity and ease of development. Since the target machine is at a design state, the simulation studies performed assisted the refining of the machine's design. The modularisation of code allowed changes to one component of the simulator to have no effect on the others.

The simulator uses message passing to simulate the circuits in the machine. It was developed and implemented on an ^IBM-XT using *PORT.

*PORT is a Trade Mark of University of Waterloo
^IBM is a Trade Mark of International Business Machines

ACKNOWLEDGEMENT

This work would not have been possible without the support and guidance of my supervisor Mr. Gary J. Stafford. The initial idea for the new machine was initiated by Mr. Stafford. The facilities provided by the University of Wollongong Computing Science Department and IBM Australia were extremely useful. Many thanks are extended to Ms Elisabeth Hilton and Mr. Kingsley Tisseverasinghe whose generosity and scholarship have saved me from many errors of omission and commission. Finally I must acknowledge the unfailing encouragement and careful comments of my wife Caroline and all the fellow graduate students within the Department of Computing Science.

TABLE OF CONTENTS

1. Introduction

An overview of the machine being simulated is presented to facilitate the understanding of the actual simulator. The machine is being developed at the University of Wollongong. The simulator allows the user to visually observe the working of the internal hardware of the machine.

1.1. The aim of the project

The aim of the project is the development of a flexible simulator to visually observe the execution of a machine. The simulator was necessary to facilitate testing the conceptual ideas of the new machine. It may also be used as a debugging tool for software development in the future. Since the simulator shows the contents of all the registers, the output can be used very effectively to debug code produced by a compiler. The simulator had to be written so that it could easily be changed to follow modifications to the definition of the machine. One of the most important requirements of the simulator was to display useful information on the screen. This information must help in making design changes to the hardware of the machine and help in debugging programs. The simulator must be able to link all pieces of hardware and be flexible enough so that pieces of hardware can be added or removed with ease.

The facilities available for developing the simulator were an IBM-XT running the PORT operating system. The language used to write the code is also called PORT. The PORT language is a

mix between "C" and Pascal. It also has system functions such as "send", "receive", "transfer_to" and "transfer_from" for communication between processes. The Window Manager provides all the routines necessary to implement a program driven window.

1.2. Development concept

A process begins its existence by executing the first statement of its code. All processes created by programs are given identification tags by the operating system. The difference from a procedure is that processes do not lie within the code image of a calling program.

The PORT operating system is designed to present no difficulty in the use of processes. Unlike the ^UNIX operating system, processes in PORT are very economical.

The main advantage of this simulator is that it exploits the "process" concept to facilitate modularity and ease of development. A "process" is a sequence of actions determined by its individual program and input data. The method of development made it possible to test each individual component of simulated hardware before connection to the whole system.

1.3. Introduction of actual machine

The machine being simulated has a 32 bit logical address space and a 32 bit physical address space. The machine has two different modes, kernel and user. It uses a Memory Management

^UNIX is the Trade Mark of Bell Laboratories

Unit(MMU) based on a segmented memory addressing system using Base, Limit and Permission registers. MMU registers are at locations 0 - $5F of the logical address space when the machine is in kernel mode.

The Main features of the new machine are:

i) Multi-Processor

ii) Asynchronous operation

iii) High resolution screen

iv) Large Memory Size

v) No interrupts

vi) Intelligent Devices

vii) Simple

viii) Reduced instruction set

The machine uses multiple processors to spread the work load. The basic machine has four processors, the kernel processor and three user processors. Each processor has space for cache memory on board to reduce bus contention. Global memory is the common memory where information about all programs are stored. The memory available in the machine is of the order of eight billion bytes. The machine has a basic set of 16 registers.

The intricate details of each piece of hardware and instructions in the new machine will be presented in the discussion of

each piece of simulated hardware. The thesis is organised so that the detailed discussion and explanation of the machine is discussed where the implementation of each piece of simulated hardware is discussed. This enables the reader to relate the implementation with the actual design of the machine.

2. Method of Approach

A decision was made to use the concept of "process" to write the code for the simulator. This allows the programmer to write the code for each simulated piece of hardware separately. The alternative was to write the whole simulator as one large program. Had that been done, making changes would have been difficult due to the interactions between the pieces of code simulating different parts of the machine. This method would not facilitate step by step development of the machine or separation of concerns.

2.1. Discussion of the whole system

The machine being simulated consists of a set of hardware pieces. All the hardware components to be simulated are listed below.

1. Arithmetic Logical Unit (ALU)
2. Memory Management Unit (MMU)
3. Bus
4. Memory (MEM)
5. On Board Switch (OBS)
6. Gofer (GOF) (Connections)
7. Serial Line (SL)

These are represented in the simulator by processes. Each process has an individual task to perform. Sent and received

messages represent the interconnection lines in the real machine. The following describes the fields in the message template. More details of the fields will be given when needed.

ADDRESS	{a 32 bit number}
DATA	{a 32 bit number}
WIDTH	{2 bits indicate 1,2,3 or 4 bytes for data}
MODE	{contains 1 bit, defines kernel or user mode}
ACCESS	{uses 1 bit to indicate reading or writing}
REQUEST	{defines the type of request made by a process}

Since each process has a well defined task to perform, the code for each process was tested by using simple programs to interact with the new process created. Testing was carried out whenever the code for one of the processes was nearing completion. This facilitated the development because processes could be tested before completing the whole simulator.

One major problem in setting up the communication path is the incorrect message passing of one or more processes.

This manifests itself in one of two ways. When a process receives a message it has to acknowledge the receipt of that message. Until the acknowledgement is received the sender is reply blocked. Reply blocked processes can be detected by looking at the status of each process. Thus incorrect communication is easy

to detect and fix. This is analogous to incorrect protocol in asynchronous circuits.

Incorrect messages are more difficult to detect. By monitoring the contents of each message many bugs were detected and fixed. The bugs were varied, some processes were sending the wrong contents, and some were sending the information to the wrong process. These are analogous to wiring errors in physical hardware.

2.2. Advantages of a multiple process system

The problems encountered were only minor compared to those that would have occurred if the "process" concept had not been used. The use of multiple processes enforces code modularity and ease of development.

Using a separate process to simulate one piece of hardware enables the programmer to consider only the code for which a change was made. Otherwise the complete simulator would have had to be considered each time a change was made to any part of the code.

2.3. Role of the processes

Each unit of simulated hardware has a role to play in the system's execution. A brief description of their functions is given below:-

i) ALU - Executes instructions.

ii) MMU - Translates addresses.

iii) MEM - Stores values.

iv) Bus - Traps addresses for its board, connects a single board to global machine communication path.

v) OBS - Diverts messages to appropriate places.

vi) GOF - Carries messages

When all of the processes are connected and the simulator is working, sample programs can be executed using the simulator to monitor working of the hardware components. For example the simulator can be used to monitor how subroutine calls are done.

2.4. Starting the simulator

The simulator is designed so that it can be configured using a text file. This reduces the need to modify the actual code in the programs when design changes are made. The processes are created by a program called "Set_machine". This program is used to read the information from a text file, start the processes and set up the communication paths.

2.4.1. The contents of the text file

Each line of text consisted of the following information.

i) Name
ii) Process Number
iii) Sizes
iv) In_Ids {pronounced "in eye-dees"}
v) Out_Ids

2.4.1.1. Name

The PORT language provides support for building systems of multiple sequential processes which communicate by message passing. The "Name" provided in the text file indicates which program the process should execute.

2.4.1.2. Process Number

This is the number given to a process by the programmer to identify the process. This number is a unique number within the file. When the simulator is running, the operating system provides every process with an identification tag. The number given by the programmer is used by the program "Set_machine" to identify the different processes it has to deal with. Note that the name taken from the file does not provide a unique identification since multiple processes can execute the same program.

2.4.1.3. Sizes

The sizes refer to the address range to which that particular process can refer. For example a memory process given the sizes $0 and $3FFF will be responsible for the first 16384 physical memory locations.

2.4.1.4. In Ids

The In_Ids are the process numbers to which a particular process can listen (receive messages from). These In_Ids are the process numbers specified by the programmer. This is one of

the main reasons the programmer has to identify processes by numbers.

2.4.1.5. Out Ids

These are similar to In_Ids except that these numbers refer to the processes to which a particular process can talk (send messages).

2.4.2. A sample from the text file

The following is an extract from the text file which is used to set up the machine. Refer to Appendix III for a complete listing of a text file.

```
0  #me/Frontpanel  $0          $0              ; 1 2 3 4 5
1  #me/Alu         $0          $0              ; 2 0
2  #me/Mmu         $0          $0          1 7 ; 8 0
3  #me/Mem         $80000000   $80003FFF   9  ; 10 0
4  #me/Bus         $80000000   $80003FFF   11 ; 12 13
```

In the above example the numbers in the first column are the process numbers given by the programmer, "#me/Name" is where the code for the processes are stored. The hex values are the range of addresses each process will respond to; it should be noted that only the Memory and Bus processes use the address ranges. The numbers after the address range are the In_ids and the Out_ids. The ";" separates In_ids from the Out_ids. So for

example, #2 Mmu receives messages from #1 Alu and #7, sends messages to #8 and #0 Front panel (#7 and #8 are Gofers). All the details in the text file are analogous to a wiring diagram which explains how to make the connections between pieces of hardware in a machine.

An alternative method of setting up the simulator would be to include all the necessary details in the code of each process. This could be very tedious when changes have to be made to the configuration. If this information is included in the code and a change is required, then the actual code file would have to be changed every time. Changing the code file is time wasting since each changed file has to be recompiled before running the simulator. As well, two processes could not have used one code file.

A third possibility would be to include all the start up details of all the processes in the code of Set_machine. This method would also require modifications to a code file to change the configuration. This also implies that the parameters that define the behaviour of processes are scattered around the program Set_machine, which makes it difficult to keep track of all the information.

2.5. Program "Set machine"

This program reads the data and creates the simulator processes and informs them of the other processes with which they communicate. All the required processes have to be created before it is possible to inform each process which it has to com-

municate with. This is necessary because the process numbers given by the user in the text file are not the real identification numbers(ids). Only after each process is created is the real "id" available. Therefore the whole file of information had to be read and stored in the data structure discussed below.

2.5.1. The data structure used

The program "Set_machine" must use a data structure to match the input text. The data structure must also be flexible to handle expansions if needed. The program uses a template called "Structure" which is defined below to store all the information in memory. An array of such structures is used to store the information about each process.

```
LOW        : unsigned[32]
HI         : unsigned[32]
ID         : Pid          {pronounced "Pee eye-dee"}
IN[256]    : unsigned
OUT[256]   : unsigned
PATH_NAME  : &char
```

"LOW" and "HI" are 32 bit unsigned numbers containing the address range for that process. The "LOW", "HIGH" and "PATH_NAME" were stored only for debugging purposes. "ID" is the process-id given by the Operating System when created using the function "Create". Type "Pid" is a system defined type. "IN[256]" and "OUT[256]" are vectors to hold the values of In_Ids and Out_Ids. A vector of 256 has been allocated so that the memory will be more than sufficient for future additions. The "PATH NAME" refers to the name of the code file.

A question may arise that the data structure uses a lot of memory. The usage of memory at this point is not very important because the program "Set_machine" terminates after starting up the simulator. Therefore the execution of the simulator is not affected by the program "Set_machine".

2.5.2. Using the data to start up the machine

The program sets up a table of the above mentioned data structures. The table is indexed by the process number in the text file given by the programmer.

The following algorithm explains the creation of each process.

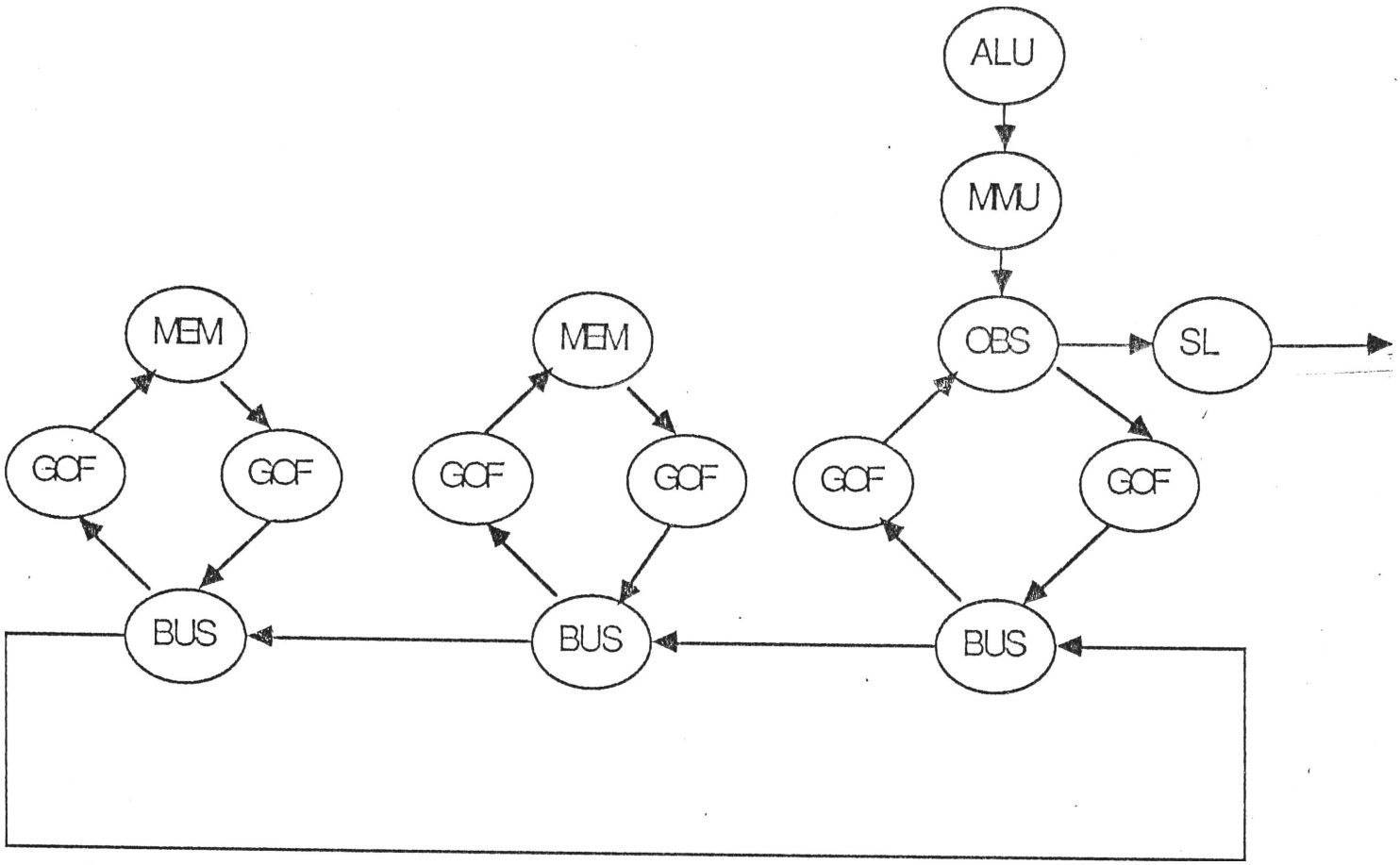
```
while not end of text file
        read a line of text
        pick record using argument 0
        Create a process using argument 1
        remember its process-id in ID
        store the sizes using arguments 2 & 3
        send the sizes to the process created
        while the next argument is not a ";"
             store the argument value in In_Ids[]
        while there are more arguments
             store the argument value in Out_Ids[]
for each process entry in the Structure table
        while there are more In_ids
             send the real In_ids to this process
             send invalid id to indicate end of Ids.
        while there are more Out_ids
             send the real Out_ids to this process
             send invalid id to indicate end of Ids.
```

As can be seen, the configuration of the machine can be modified very easily. When a new piece of hardware has to be simulated, the text file can be changed to accommodate the new addition.

3. Details of simulated hardware

This chapter will discuss the important details of the machine components where relevant and the implementation details of each simulated piece of hardware. The multiple process environment paves the way for convenient coding of processes. It allows the programmer to write the code for each process separately and test it before linking it with the main simulator. For example the instruction definitions can be modified and tested using the simulator. In this section more details of the machine will be discussed before starting the discussion on implementation. All the definitions of the machine features are not part of writing the simulator or part of the project therefore justification for every feature of the machine will not be discussed.

When using multiple processes testing of one process requires the processes with which it communicates. Therefore all processes which were not completed at a particular stage were replaced by dummy processes for the purpose of testing. At this stage the dummy processes do not assume their eventual role, but act as communication relayers. The following is a diagramatic view of the communication path. The arrows indicate the direction of the communication.



1 - P18 Represent the Communication Path

Figure 3.0 Communication Path

Testing individual processes before linking them to the main system helps a great deal when debugging. The processes however can only be partially tested before linking them with the simulator, but it helps to identify communication errors. This can be also used very effectively to test for correctness of functions. The details of the MMU will be first discussed because the MMU was the first program to be implemented and tested.

3.1. Introduction of the MMU

The main function of the MMU is translation of logical addresses into physical addresses. The MMU also checks for invalid addresses and invalid operations on addresses.

3.2. Positioning the MMU within the environment

The MMU sits between the bus and the instruction processor. It receives requests from the instruction processor. Then the MMU does the necessary address translations and error checking. If the MMU is satisfied it sends the request to the bus and waits for a response. After the response is received, it then replies to the instruction processor with the appropriate details.

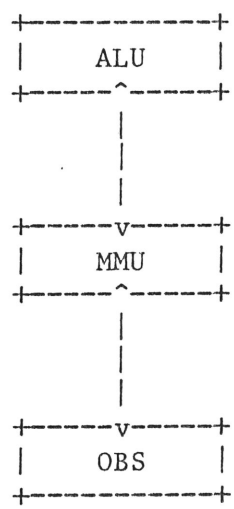


Fig 3.1 Diagram of MMU's position within the whole machine

3.3. Using the message template fields within the MMU

The message template contains 6 different fields, each with a different purpose. The following paragraphs will describe only

five of the relevant fields in detail because the "REQUEST" field was not used by the MMU process

The most significant bit in the ADDRESS contains information to decide which half of the segment set to use. If the most significant bit was on, then the upper half (i.e. 24 - 47) should be used and if the bit was off then the lower half (i.e. 0 - 23) should be used. After the decision on which half to use has been made, then the next most significant bit that is on indicates which segment to use. All the other lower bits give the value of the offset into this particular segment.

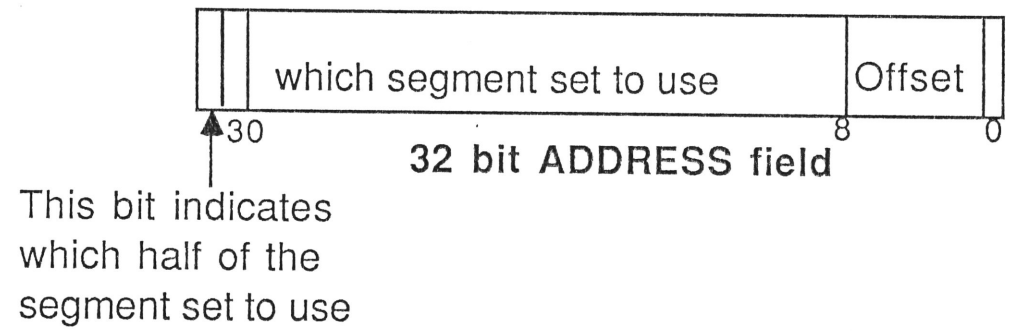


Fig 3.2 Example of 32 bit ADDRESS

The MODE field indicates if the process is running under kernel or non-kernel mode. This distinction is very important

for the MMU because it does not do any address translations when the machine is running in kernel mode.

The ACCESS field indicates a read or write access. This is very necessary for permission checking. ACCESS is a one bit field (1 - read and 0 - write).

The WIDTH field gives the width of the data. This field is also used to indicate error conditions when returning a message. The WIDTH field will contain an ok signal or an error condition.

When the data is addressed to the MMU, the 32 bit DATA value presented to the MMU contains information about the Base, Limit and Permission registers. All this information is within the 32 bits that have been provided for the DATA field.

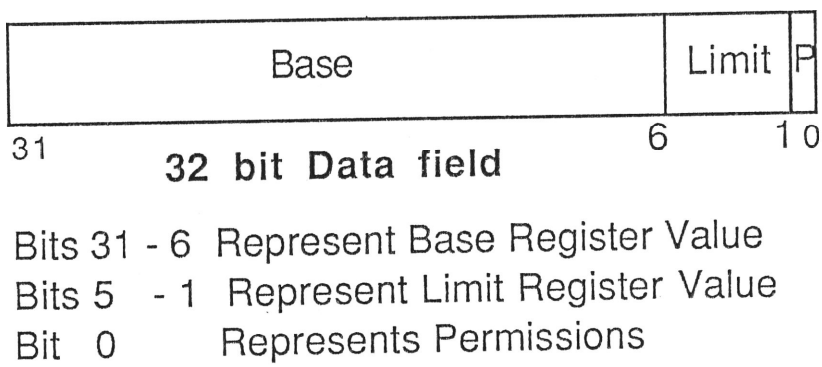


Fig 3.3 Diagram of DATA field

3.4. Design of the MMU simulator

The main function of the MMU is to repeatedly receive messages from the ALU, translate the logical addresses to physical addresses, send the address along the communication path and reply to the ALU.

The memory management facility works in one of two modes. They are :

(1) Kernel mode

(2) User mode (non-kernel)

When the kernel of the Operating System is running, the memory management is effectively short circuited and the address given by the kernel is taken as the physical address.

The MMU is most useful when a non-kernel process is running. When in non-kernel mode the MMU has to do the mapping of addresses and trapping of invalid addresses. It also has to check for permissions.

The 32 bit address presented to the MMU is used to select one of 48 sets of registers. Each set of registers consists of a Base register, Limit register and a Permissions register. Assuming the bits are numbered from 0 to 31 from least significant to most significant, then the question of which of the 48 segments is to be used is determined from the bits 31 to 8.

When an address is passed to the MMU two types of errors can occur: a limit error or an access error. Limit errors occur when the processor tries to refer to an address outside the specified address space. An access error occurs when an attempt is made to write to a read only segment. An access error can also occur when an attempt is made to read from a segment for which read access is denied. Checking for these errors is quite simple.

The WIDTH field of the message is used to indicate the type of error. The two bits in the WIDTH field are used for this purpose. The WIDTH field is used because this field is redundant when replying.

3.5. Implementation of the MMU

The implementation of the MMU was done in several phases. Starting from the initial design, testing and tracing work had to be done to pick out the best and most efficient solution to the problem. The following discussion will explain the initial design, show the results that have been obtained by tracing this design and discuss the changes that were made at each stage. The trace results obtained will show the effects of these changes when executing the program. Tracing procedures carried out will be discussed in detail. All other major code modules were traced and refined in a similar manner. The MMU was the first process implemented and traced, so the tracing mechanism used and the results obtained will be discussed here.

3.5.1. Phase 1

A function "Mmu" was written to repeatedly receive messages from a process. When a message is received from a process, it is passed on to a function called "Handle_msg" which deals with part of the contents of the message received.

The Handle_msg function has two major sections. If the MODE of the message is "kernel", then Handle_msg will continue processing the functions needed for kernel mode addresses. If the message is not in kernel mode then the processing will execute the functions needed for non-kernel addresses. The following is a pseudo code version of Handle_msg:

```
Accept a message from function Mmu
if kernel mode
        process message in kernel mode
else
        process message in non-kernel mode
```

3.5.1.1. Processing a message in kernel mode

Assuming the MODE is "kernel", then the Handle_msg function should check if the address is between 0 and 96 inclusive, and if the access is "write". If both the conditions are true then the function "Set_reg" is executed. The Set_reg function uses the DATA field of the message that was passed in from Handle_msg to set the registers. All the information necessary to set the registers are encoded into the DATA field of the message. Set_reg sets the Base, Limit and Permission registers. It uses

the ADDRESS field to work out which register set to use. After a particular register set has been assigned values, all the sets with a higher number are made inaccessible. To make registers inaccessible, a value of 0 is assigned to the Permission register.

When the kernel wants to read values from memory, then the function "Access_memory" is invoked. This function will pass the address directly to memory because the request came from the kernel. No address translations are done in kernel mode.

3.5.1.2. Processing a message in user mode

When a process is running in user mode then all the necessary address translation and error checking has to be done.

3.5.1.3. Converting virtual addresses to physical addresses

Access_memory converts a logical address to a physical address. This function performs a very important role in the Mmu. In address translations, the register set the address refers to is identified. Which of the 48 sets to use is determined from the most significant non-zero bit (ignoring the highest bit) of the address. This can be done in many different ways. In phase1 two tables called "Mask_table" and "High_bit_numbers" are used to determine the register set and the offset.

A loop was constructed to shift the 32 bit address by 8 bits each time around the loop to find out which byte contained the

necessary bit to determine the register set. This method was used instead of 32 one bit shifts so as to reduce the amount of testing that would have to be done to determine which bit was on. The highest bit that was turned on is located by indexing into the table of High_bit_numbers. On obtaining the highest bit that was turned on within the 8 bit byte, the number of bits that have been shifted can be added to work out the highest bit relative to the 32 bit address. This value is used to index into the Mask_table so that the offset can be selected.

The High_bit_numbers table is indexed by an 8 bit number, therefore this table contains 256 locations. When the table is indexed by a number (0 <= number < 256) it returns the position of the most significant bit that was on.

For example consider the following address:

$00003543

When the High_bit_numbers table is indexed with the value $35, the value 6 is obtained, this being the most significant bit that was on, (if the bits are numbered 1 - 8 from right to left.)

The Mask_table contains mask values to obtain the offset into a particular segment. The values in this table mask off the the most significant bit that was on, which indicated the register set to be used. The table is indexed by a value between 0 and 23 inclusive and therefore contains 24 locations. The mask

values are worked out assuming the address was shifted right by eight, so that only the upper 24 bits are taken into consideration.

eg : value of 6

If the Mask_table is indexed with the value of 6 then the value of $0000001f results and this will mask off the bit which was used to indicate the register set. After masking off that bit, the offset into this particular segment to be used is obtained. Thus for $3543, the register set is 6 and the offset into that segment is $1543

3.5.1.4. Obtaining the register values

After calculating the register set to use, the Base, Limit and the Permissions are obtained by indexing into the register tables. Summing the offset and the Base value gives the physical address. Before summing the values a check is made for a possible limit error. After obtaining the physical address, the access permissions must be checked for any violations. If a limit error or an access error occurred, then an error indication is returned.

3.5.2. Tracing of Phase 1

The execution of the MMU was traced to ascertain where it was spending most of the time, and to find out where changes could be made to make the running of the MMU more efficient. The main concern was to cut down on the number of instructions the

MMU took to complete one translation cycle. A file of addresses and data was created to simulate input from the ALU. The file contained 500 sets of data. The same file was used throughout all the testing and tracing phases.

An example of a set of data is as follows:

uwf $1234 $678

"uwf" - user mode, write, full word

$1234 - Address field

$678 - Data field

Trace Results of Phase 1, No. of instructions per function

Mmu	20393
Handle_msg	19298
Access_memory	164340
Set_reg	19851
Total*	247815

*Total includes instructions from other functions.

After the MMU function was traced with the sample data, the areas that had to be changed were self evident. Analysing the trace by function output, it was found that the program was doing the bulk of its work in Access_memory. Analysing the trace by instruction output it was noticed that most of the instructions were executed to pass the message to, and return the message from, functions that needed the message. The initial intention

was to reduce this unnecessary work load of passing the message to each function.

When the records have to be passed to a function the whole record must be copied each time it is passed. Copying records consumes a lot of instructions. Copying a record each time a function needs it is unnecessary when it can be set up so all functions have access to it.

3.5.3. Phase 2

The problem of passing the message was affecting all the functions, so a decision was made to have the message as an external so that all the functions have access to the message without having to pass it to each of the functions that used it. This method was decided upon after considering the option of passing a pointer to the message. The pointer passing method was not very efficient because the processes still had to execute unnecessary instructions to pass a pointer to the message. The biggest cost with passing a pointer to a record occurs when a field in the record has to be accessed. When a field within the record is needed by a function, the address has to be computed before accessing the field.

Trace Results of Phase 2, No. of instructions per function

Mmu	8909
Handle_msg	2430
Access_memory	157033
Set_reg	19851
Total*	212132

*Total includes instructions from other functions.

After tracing the execution of phase2 it was distinctly noticeable that a large number of instructions had to be executed to do 32 bit shifts. This was mainly due to the fact the the 8086 processor performs 32 bit shifts by only shifting a single bit at a time. So multiple shifts have to be done in a loop. The Access_memory function had to do many 32 bit shifts to work out the high-bit that was on in the 32 bit address given to Access_memory.

The main aim of Phase 3 was to reduce the total number of instructions executed by reducing the number of 32 bit shifts.

3.5.4. Phase 3

Changes to the function Access_memory were centered on where 32 bit shifts occurred. In the previous phases there was a great deal of unnecessary shifting because 8 bits were shifted each time around the loop to find out if the bit sought occurred in those 8 bits. One half of the search area was eliminated by

doing a simple check which did not require any shifting of bits. The pseudo-code for this particular check is as follows.

```
if (Address bit-wise and with $FFFF0000)
     upper 16 bits
else
     lower 16 bits
```

From the above check it was possible to decide which half the high bit was in. Only the bits 16-8 from the lower 16 bits have to be considered because only 24 bits need to be used. After it has been decided which half to look at, then a test value and a mask value have to be created to check the address to find out the highest bit that was on. The test value is used to check a particular bit to see if that bit was on, the mask value is used to find out the offset into the particular segment. This method is basically a linear search (i.e. each bit starting from the most significant bit is tested with a new test value each time down to the least significant bit in that particular half.)

The pseudo version of this check is as follows.

```
if in upper 16 bits
    initialise Test
    initialise Mask
    if (Address bit-wise and(&) with Test)
        work out register set
        work out offset
    else
        shift Test value right by 1
        shift Mask value right by 1

else in lower 16 bits
    initialise Test
    initialise Mask
    if (Address bit-wise and(&) with Test)
        work out register set
        work out offset
    else
        shift Test value right by 1
        shift Mask value right by 1
```

A loop was created to check each bit consecutively. The starting mask value depends on the starting test value. If a decision was made to search the upper half, then the starting test value will be $80000000 and the starting mask value will be $7FFFFFFF. The test and the mask values will change according to the search area.

The following trace results were obtained after making changes.

Trace Results of Phase 3, No. of instructions per function	
Mmu	8909
Handle_msg	2430
Access_memory	94699
Set_reg	19851
Total*	148906

*Total includes instructions from other functions.

3.5.5. Phase 4

This phase of modification uses a similar technique to that of phase3. It is that of reducing the search area by doing a bit-wise "and" to check if any bits were on in a particular area of the address field. This method is very tedious to write but it should reduce the number of instructions needed to find the high-bit number. The technique to be used in Phase 4 is a binary search technique, which eliminates one half of the search area after each comparison.

The code for searching the high bit number using a binary search technique was modularised to its extreme. It was written so that after each check a different function was called. An example of such a test was mentioned in Phase 3. The technique of testing and eliminating the search area was used in all the functions until the appropriate bit was found. This method was rather complex to implement because of the number of functions that had to be used. The high number of functions were needed to

cover all possibilities when doing a binary search. The organisation of the functions were similar to a binary tree.

Considering the function calls as a binary tree, each leaf node contained a mask value and a high-bit number. The values used for mask value and the high-bit number were assigned to external variables. This allows all functions free access to these values. The only drawback to the binary search technique is the amount of manual work needed to create the functions. Phase 3 needed the least amount of manual work. This was possible because only the initial values of Test and Mask had to be assigned. In phase4 all the possible Test and Mask values had to be worked out manually. In phase 4, the 32 bit shifts have been eliminated.

The "Set_reg" function used a loop to assign zeros to all register sets not in use. It was changed to use the system function "Zero" which made it more efficient.

Trace Results of Phase 4, No. of instructions per function	
Mmu	8909
Handle_msg	2430
Access_memory	41201
Set_reg	4210
Total*	98911

*Total includes instructions from other functions.

From the trace results it can be seen that there has been a considerable saving in the number of instructions consumed by Access_memory.

After analysing the trace results more thoroughly it was seen that more savings were possible by reducing the number of functions. This was done by bringing the setting of mask and high bit numbers functions up by a few levels. Savings could then be made in the number of instructions needed to make function calls. In the PORT language this is not a very big saving (only uses 2 instructions per call and return), but in a language such as 'C' there would be a much bigger saving.

After all the major changes were done, some minor adjustments were done to reduce the number of variables used. This was basically a clean-up process to reduce redundancy. The trace results of the final version follows.

Trace Results of final, No. of instructions per function	
Mmu	8909
Handle_msg	2430
Access_memory	29107
Set_reg	4210
Total*	86818

*Total includes instructions from other functions.

3.5.6. Advantages of tracing

From the above results it is obvious how advantageous it is to monitor and make programs more efficient. The number of instructions needed to simulate all the MMU operations have been reduced from 247815 down to 86818 which is a very large saving. Since the number of instructions executed have been reduced, the speed of execution has been increased.

3.6. Arithmetic Logical Unit (ALU)

The main function of the ALU is the execution of instructions. It is also known as the Instruction Processor (IP). The ALU talks to the MMU when it has to read an instruction or when it has to make a reference to data memory. When the ALU needs to display any information it sends a message to the Front Panel to display the information on the screen. This is not part of the hardware definition but rather visualising execution of test programs. The major parts of the machine that relate to the ALU will be discussed in detail.

3.6.1. Registers

The machine has a basic set of 16 thirty two bit registers. These registers can be considered in two parts: 8 registers which are manipulated directly by the program which is defined by those registers, and another 8, which are hidden from the program but used for other purposes by the ALU. The machine has two sets of registers, one for kernel and the other for user processes.

Since the machine has two sets of registers the simulator had to be written so that values of these registers can be stored and made available at all times. This could have been done in many different ways such as linked lists, structures, 32 variables or vectors. The simulator utilises a vector to hold the active 16 registers that exist in the machine. This method has been chosen because vectors can be indexed by using values extracted from instructions. All the register values are displayed on the screen when the machine is running. More details of the displays will be discussed in the front panel section. The following table lists all 16 registers as they have been defined.

Register Name	Used for
General Register 0	General purpose
General Register 1	General purpose
General Register 2	General purpose
General Register 3	General purpose
General Register 4	General purpose
Frame Pointer	Pointer to Local data Space
Old Frame Pointer	Old value of Frame Pointer
Program Counter or Zero	Points to the next instruction treated as zero in ALU
Old Program Counter	Old value of Program Counter
Status Register	Flags affecting Execution
History Register 0	Contains an Old Program Ctr.
History Register 1	Contains an Old Program Ctr.
History Register 2	Contains an Old Program Ctr.
History Register 3	Contains an Old Program Ctr.
History Register 4	Contains an Old Program Ctr.
History Register 5	Contains an Old Program Ctr.

Figure 3.4 Register Set

The five general purpose registers are used for all arithmetic and logical operations in the machine. All operations are conducted with 32 bits of precision. These five registers can be augmented by the next two, which are the frame pointer(fp) and old frame pointer (ofp) registers. The only thing which differentiates these two registers from the five general purpose registers is the affect on their contents when certain instructions are executed, hence they are named differently to distinguish them from the other five.

The last of the first eight registers is viewed in two different ways. When instructions are being fetched from memory this register is treated as a pointer into the area of memory which contains instructions. It is treated in the same way for instructions which change the location of execution such as jumps and calls. However, for all other instructions this register presents another appearance. It is viewed as a register which contains the value zero. Thus, for example, it is not possible to change the program counter (pc) with arithmetic instructions. Since this restriction applied to the (pc) register, the simulator had to be written to cope with this problem in a simple manner.

Now that the first eight registers have been discussed, the second eight registers will be discussed. The old program counter (opc) is used with call and return instructions to provide a fast subroutine call mechanism.

The status register contains various bits. The main information represented by the bits are:

a) Whether the process is in kernel or user mode

b) The reason for switching to kernel mode from user mode.

c) Whether to simulate a switch state instruction.

d) The one condition code of the machine which is the carry bit

The last six registers are history registers, which are changed by instructions such as jump and call. To see how these registers are manipulated, a simple algorithm can be looked at. The assumptions are that the "Destination" has been defined, either by computing the address from the short version of the instruction and the current program counter, or by loading the 32 bit value which followed the jump instruction.

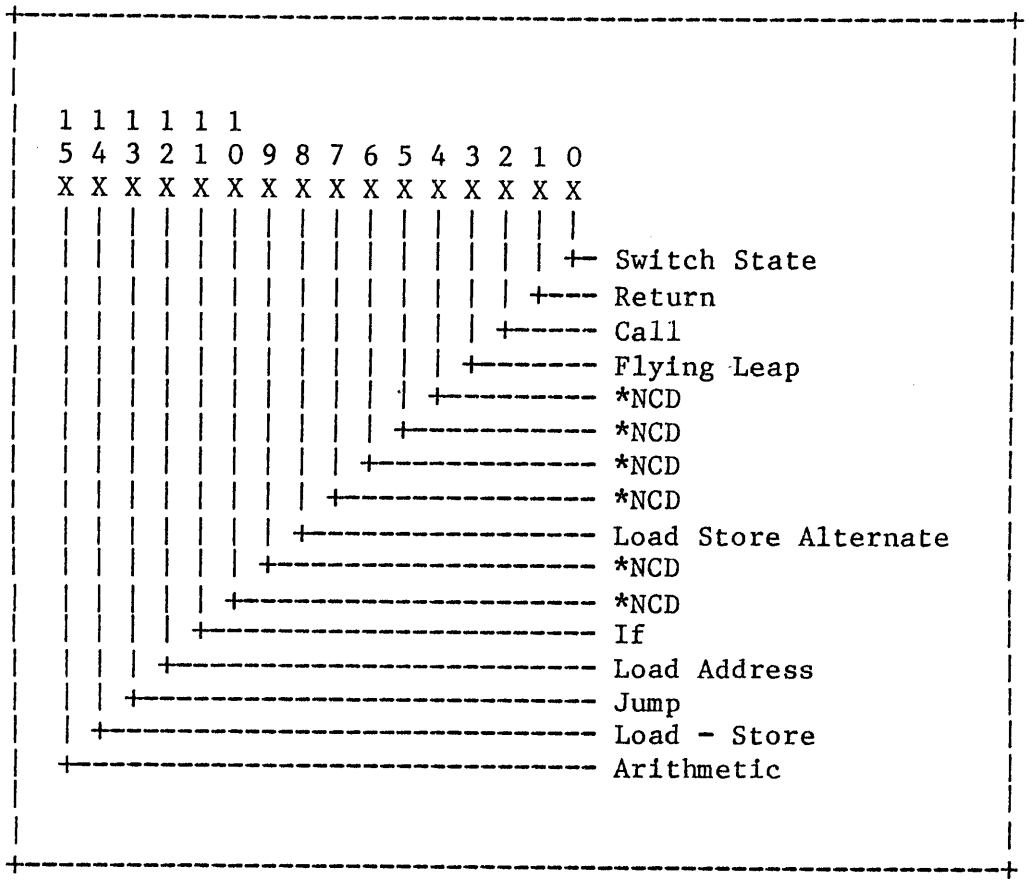
```
Register[HR5] = Register[HR4]
Register[HR4] = Register[HR3]
Register[HR3] = Register[HR2]
Register[HR2] = Register[HR1]
Register[HR1] = Register[HR0]
Register[PC]  = Destination
```

Figure 3.5 History register Algorithm

The History registers which are a part of the 16 registers could have been implemented using a circular list, but it was implemented utilising an array to hold the values. Using a circular list would have made jumps and calls simpler. The array method was used for the sake of simplicity and code modularity. Using a circular list in this case would have made other operations more complex.

3.6.2. Instruction Set

There are 16 basic instructions in this machine. All instructions are an integral number of half-words in length. Which of the 16 instructions to perform is indicated by the most significant non-zero bit of the instruction packet. The following diagram gives the basic instructions and specifies which bit indicates which instruction.



*NCD - Not Currently Defined

Figure 3.6 Basic Instructions

Since the most significant bit (MSB) indicates which instruction to execute, the simulator had to implement a very efficient means of obtaining the MSB from a given instruction packet. Therefore it was decided to write a function to strip the bits in the instruction and return the MSB that is on.

In the following paragraphs, the details of all the instructions implemented will be discussed.

3.6.3. Arithmetic

All arithmetic and logical instructions are of the form shown in the following figure.

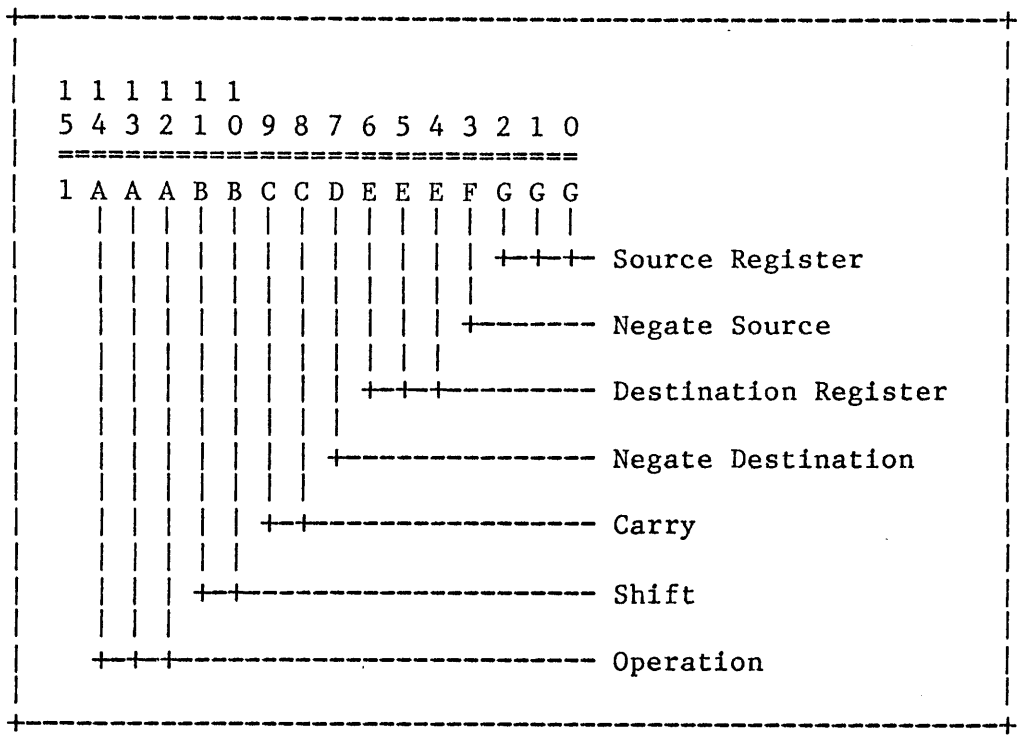


Figure 3.7 Arithmetic Instruction

These instructions operate on the contents of the seven registers, plus the register containing zero. To describe these instructions the following notation will be used. This notation will be discussed before the specific instructions are dealt with.

The notation:

Register[X]

denotes the address or contents of the register "X" as appropriate.

For convenience, in describing how the arithmetic instruction works, four pseudo registers will be defined. These are named "Sourcel", "Source2", "Carry" and "Result". These are all 33 bit registers except for "Carry" which is 1 bit.

The bits of the instruction will be referenced by their letters which were shown in figure 3.7. Note that the concatenation of bit names implies the concatenation of their respective bit values, thus AAA denotes a value from 0 to 7 as extracted from the three bits in the instruction at the location shown in the figure 3.7.

If the "D" field is on, then the destination register value is complemented, before being loaded into Sourcel. If the "F" field is on, then the source register value is complemented before being loaded into Sourcel.

The "CC" field defines the value of a local variable called "carry" used within arithmetic operations.

```
if CC == 0      carry = 0
if CC == 1      carry = 1
if CC == 2      carry = carry bit from status register
if CC == 3      carry = complement of carry bit from
                        status register
```

3.6.3.1. Operations

If AAA is equal to the value 0, the operation requested is an addition. The sum of Source1, Source2 and Carry is placed in the register Result. This will result in 33 bits of information. Since the addition is done with 33 bit registers and the 8086 processor only does addition in 32 bits the simulator had to have a special addition routine to cope with these conditions.

If AAA is equal to the value 1, then the operation requested is a bitwise "And". The bitwise "And" of Source1 and Source2 is placed in the register Result.

If AAA is equal to the value 2, the operation requested is bitwise "Or". The bitwise "Or" of Source1 and Source2 is placed in the register Result.

If AAA is equal to the value 4, then the operation is a shift to the left of the value of Source2, by the value of Source1. The Result is set equal to Source2. For each bit shifted the Carry bit is copied to the least significant bit of the Result.

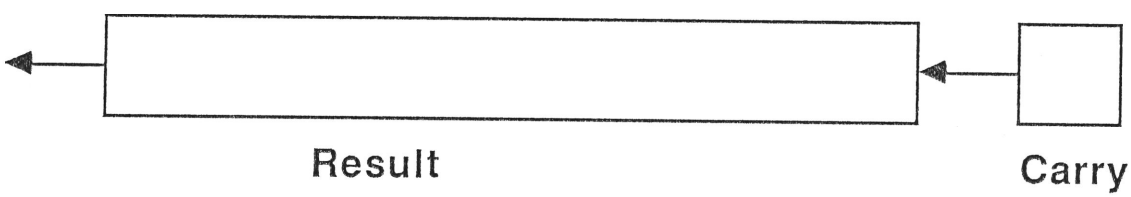


Figure 3.8 Shift Left

If the value of AAA is equal to 6, then the operation is the same as Shift Left, except that the Carry is copied to the bit which was moved from the 33rd position of the result.

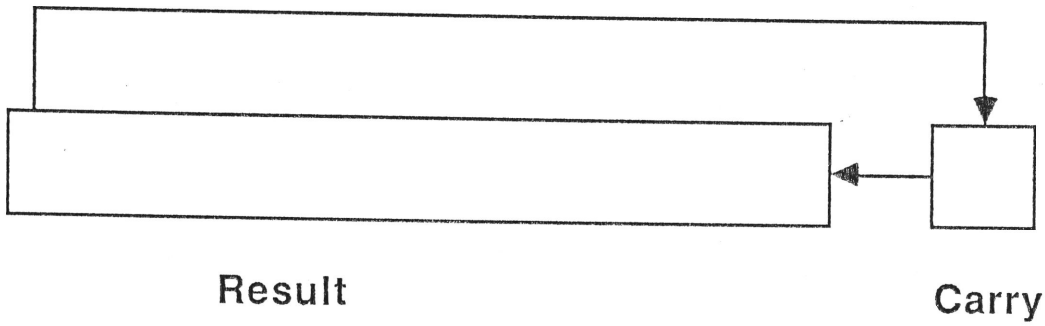


Figure 3.9 Rotate Left

If AAA is equal to the value 5, then the operation is a shift to the right of the value of Source2, by the value of Sourcel and the Result is set to Source2. For each bit that was shifted, the Carry bit is copied to the most significant bit of the Result.

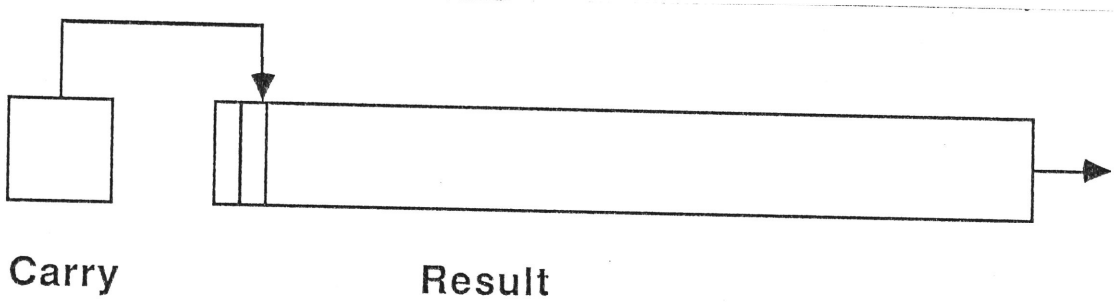


Figure 3.10 Shift Right

If the value of AAA is equal to 7, then the operation is the same as Shift Right except that the Carry is set to the bit which

was moved from the least significant position of the Result after each shift.

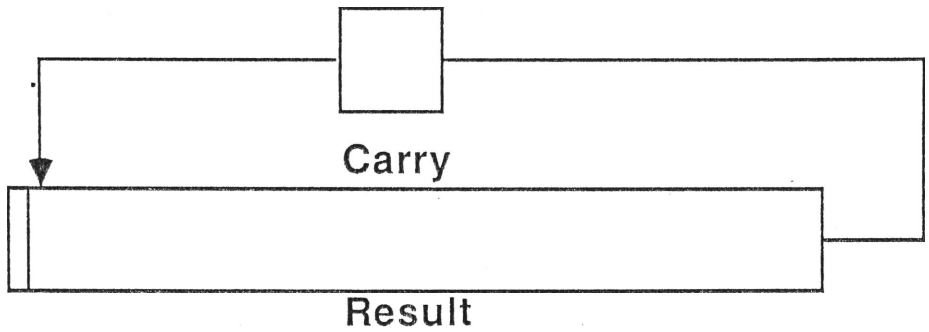


Figure 3.11 Rotate Right

3.6.3.2. The shift field

If the value of BB was 0, then the Result and Carry registers are left as they were. The Result stays the same.

If the field BB has the value 1, then the Result register is shifted left one bit and the value of the Carry register is added.

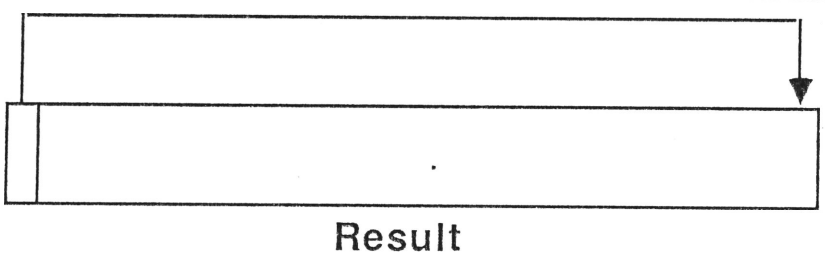


Figure 3.12 Shift left one bit

If BB has the value 2, then the Result register is shifted right by one bit and the Carry register is shifted left 31 bits and added to the result register. The Carry register is set to the value of the bit which was removed from the least significant bit of the Result register.

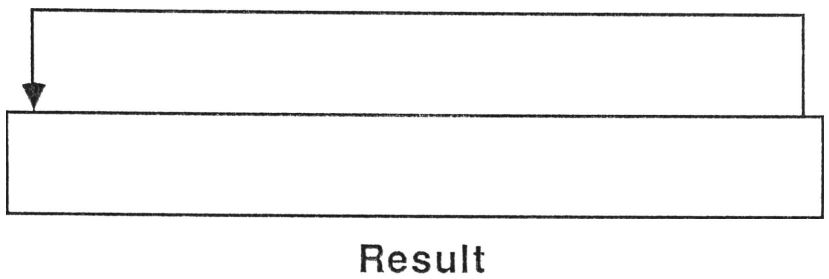


Figure 3.13 Shift right one bit

If BB has the value 3, then the Result register is shifted right by one bit and the 32nd bit is set to the 31st bit. The Carry register is set to the value of the bit which was removed from the least significant bit of the Result register.

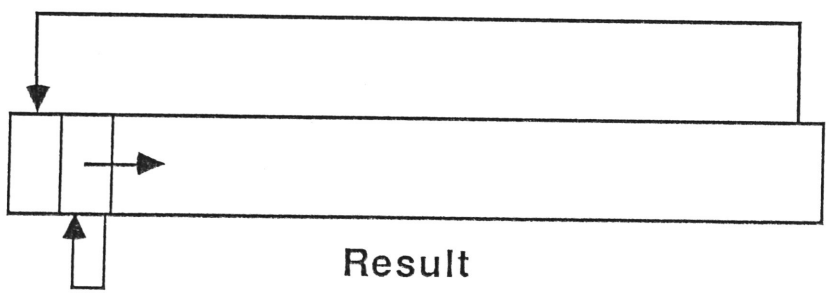


Figure 3.14 Shift right one bit

Since the maximum bits per word is 32 bits, and the arithmetic operations required 33 bits, temporary variables had to be used in the simulator to store partial values.

3.6.3.3. The destination field

The contents of the Result register are placed back into Register[EEE] and the value of the 33rd bit of the Result register is placed into the least significant bit of the status register. Since the values obtained from the EEE bits can equal 7, special precautions had to be taken when writing the simulator to avoid storing values in register 7.

3.6.4. Load Store

The Load and Store instructions are either one, two or three half words long. The extra half-words are necessary for various addressing modes and will be discussed later. The discussion of Load and Store will make reference to the following figure.

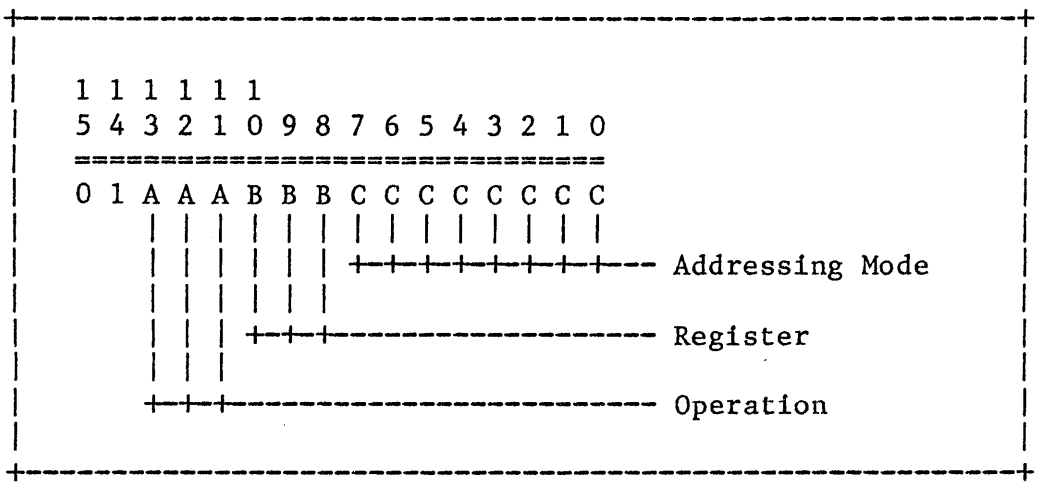


Figure 3.15 Load or Store

3.6.4.1. Field Definitions

The BBB field of the instruction specifies one of the eight registers available to the arithmetic unit. This forms either the source if the instruction is a Store, or the destination if the instruction is a Load.

The following table gives the meanings of the operation field

AAA	Operation Implied
000	Load 16 bit zero fill
001	Load 32 bit
010	Load 16 bit sign extended
011	Load 32 bit
100	Store 16 bit
101	Store 32 bit
110	Store 16 bit
111	Store 32 bit

Table 3.1 Load Store Operations

3.6.4.2. The Addressing Mode

The following figure displays the assignments of the bits in the addressing mode.

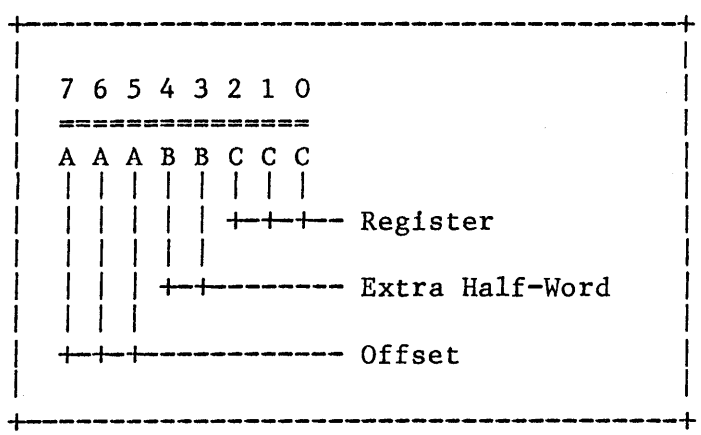


Figure 3.16 Addressing Mode

The only form of addressing specifies a register and an offset. The contents of the register are added to the offset, and the resulting value is the address of the operand. The offset can be either 3 bits, 4 bits, 16 bits, or 32 bits, depending on the value of the BB field. The following figure will explain how the offset is calculated from the bit positions. Note that the size bit of the instruction is also involved.

Instruction	Next Half-Word	Third Half-Word
01XXsXXXaaabbXXX	QQQQQQQQQQQQQQQQ	PPPPPPPPPPPPPPPP

sbb	Offset Implied
000	aaa
001	laaa
010	QQQQQQQQQQQQQQQQ
011	PPPPPPPPPPPPPPPPQQQQQQQQQQQQQQQQ
100	aaa0
101	laaa0
110	QQQQQQQQQQQQQQQQ
111	PPPPPPPPPPPPPPPPQQQQQQQQQQQQQQQQ

Figure 3.17 Implied Offsets

The addressing mode function should be able to get the correct amount of instruction packets when necessary because of the varying size of the offset.

3.6.5. Jump

The following diagram shows the bits in the Jump instruction.

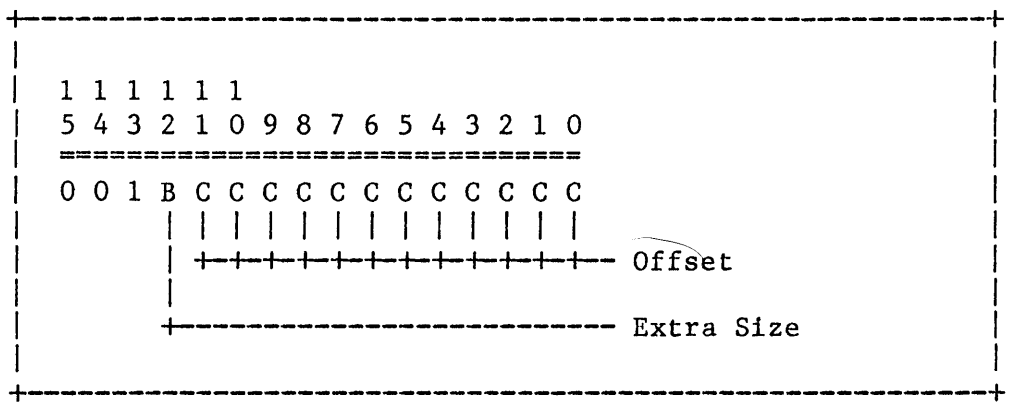


Figure 3.18 Jump instruction

If the B field has the value 1, then the destination address is contained in the two half-words which follow the jump instruction. This provides for access to any location in memory.

If the B field has the value 0, then the area of memory which is accessible is limited by the available number of bits.

This implies that when simulating the "jump" instruction, there has to be a method of obtaining more instruction packets when necessary. The history registers would also have to be assigned the correct values. When calculating the short offsets,

the 12 bits have to be sign extended and the program counter must be incremented at the right time.

3.6.6. Load Address

The Load Address instruction essentially skips the operand fetch of the Load instruction. Instead of computing the address of the memory operand, then fetching the operand, the address is treated as the operand. The following diagram shows the Load Address instruction format.

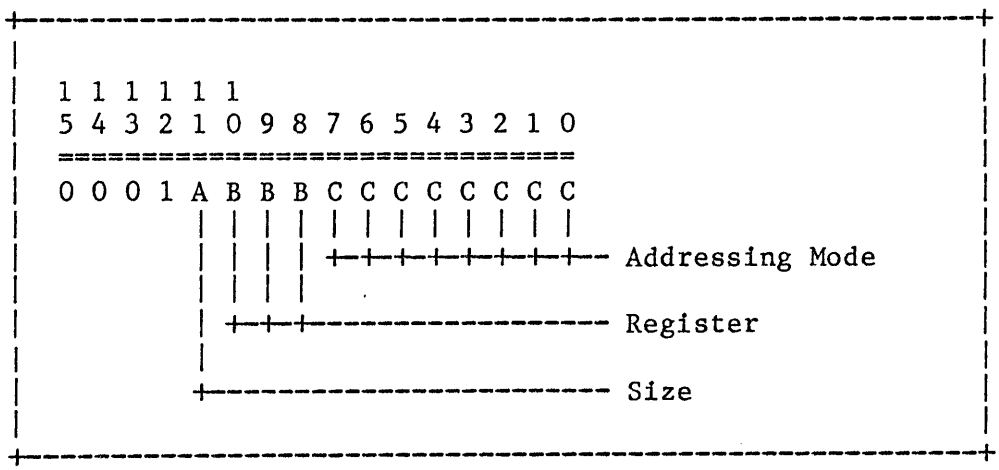


Figure 3.19 Load Address

It may be noted that the addressing mode register and size fields of this instruction are in the same bit position as in the Load and Store instructions. This makes instruction decoding for this instruction simply a subsection of the instruction decoding of the Load and Store instructions. This implies that there should be a function to work out the addressing mode.

3.6.7. If

This machine has a "If" instruction which is essentially a combination of a comparison and a conditional branch. The following figure gives the bit assignment for this instruction.

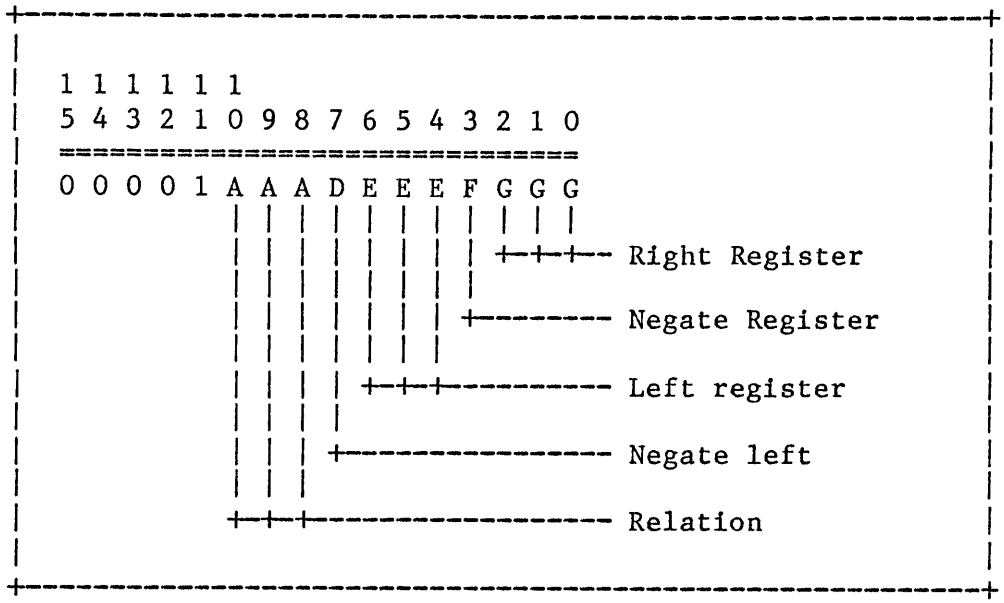


Figure 3.20 If instruction

The two operands are specified just as they are for arithmetic or logical instructions. There is a third operand found in the following half-word which is a 16 bit signed value which is to be added to the program counter if the test made is true. The test is specified in the relation field.

AAA	Do Jump if it is True that
000	Left Equal to Right
001	Carry bit is zero
010	Left Less Than Right (unsigned)
011	Left Not Greater Than Right (unsigned)
100	Left Less Than Right (signed)
101	Left Not Greater Than Right
110	Carry bit is one
111	Left Not Equal to Right

Table 3.2 Relation Definition

Since PORT supports both signed and unsigned tests, the testing of conditions will not cause much problems.

3.6.8. Access to Alternate Registers

The following figure gives the bit assignments for this instruction.

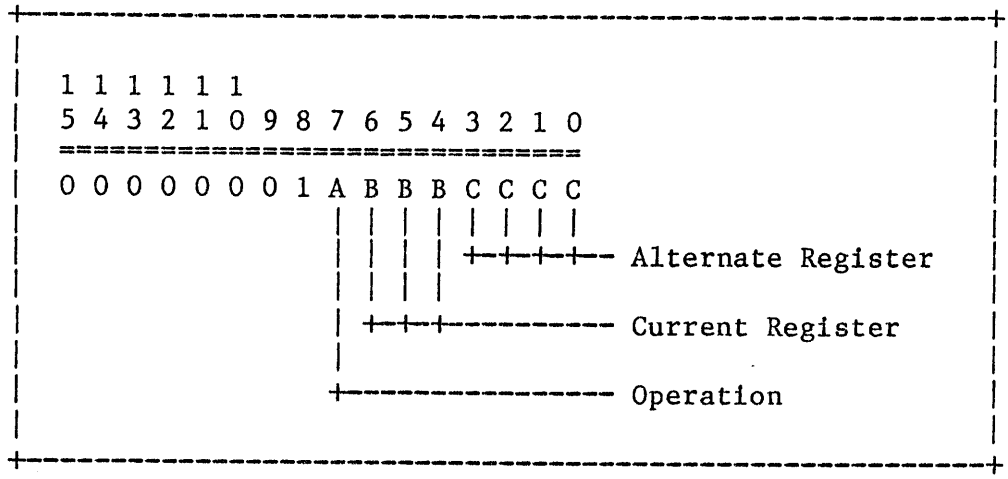


Figure 3.21 Load or Store Alternate Registers

The CCCC field indicates which of the 16 registers in the alternate set is the one in question.

The BBB field indicates which of the 8 registers of the kernel is the other register in question.

If the A field has the value 0, then the alternate register is loaded from the kernel register. If it has the value 1, then the kernel register is loaded from the alternate register. This implies that there has to be a means of storing both sets of registers, so that depending on the value of "A" the loading of registers can take place.

3.6.9. Flying Leap

The following figure gives the bit assignments for this instruction.

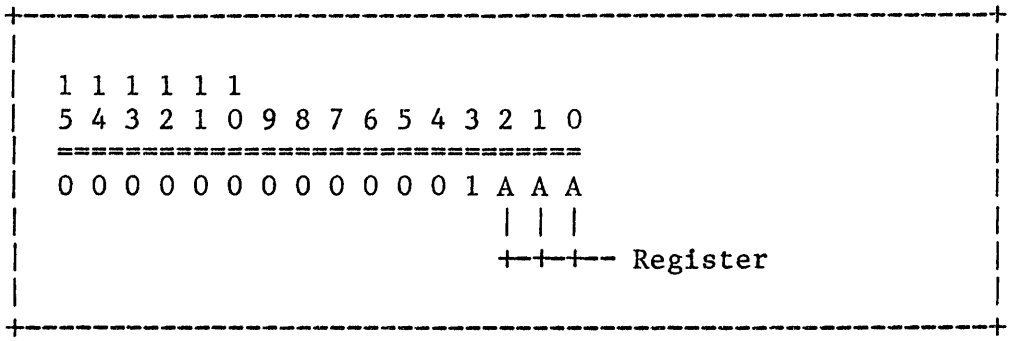


Figure 3.22 Flying Leap

The value of the destination is taken from the contents of the register specified in the AAA field. After the destination address has been determined, the same algorithm used for the Jump instruction can be used.

3.6.10. Call

The following figure gives the bit assignments for this instruction.

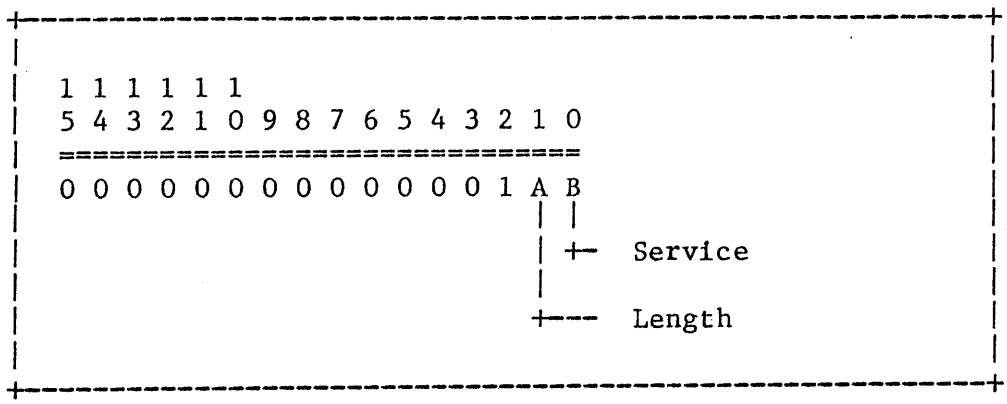


Figure 3.23 Call

The A field indicates how the destination is to be computed. If A has the value 0, then the destination is computed from the program counter and the 16 bit signed value following the instruction. If A has the value 1, then the destination is the value of the 32 bit word which follows the instruction. When the 16 bit number is used, sign extension of the number must be taken into consideration.

The B field indicates which of two different call instructions are to be used.

If B has the value 1, then this is a service call and the value of the program counter is simply stored in the old program counter register.

If B has the value 0, then the call is more complex. This requires the simulator to assign the value pointed to by the Old Frame Pointer to the Frame Pointer. Then store the Program counter at Frame Pointer + 2, and assign the value of Frame Pointer back to the Old Frame Pointer.

In both cases mentioned above, the same algorithm for saving the history registers has to be executed. This implied that the simulator needed two different functions to handle the two cases because the service call is much simpler to implement. The functions which handle the "call" instruction must have means of obtaining more instructions if necessary to work out the offset.

3.6.11. Return

The following figure gives the bit assignments for this instruction.

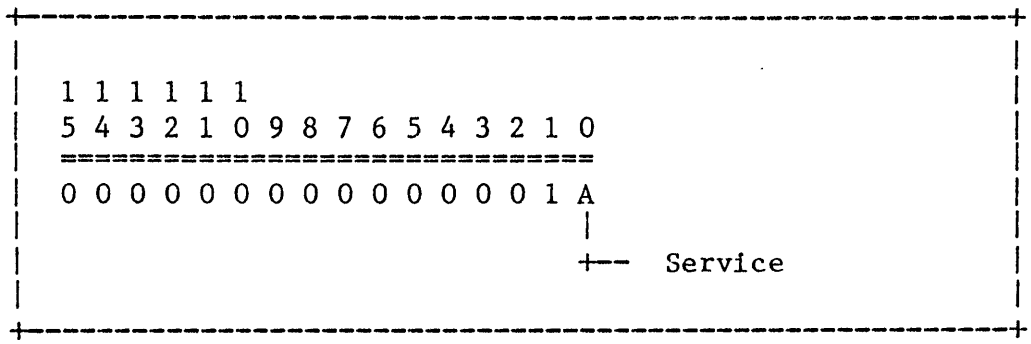


Figure 3.24 Return Instruction

If the A field has the value 1, then this is assumed to be a return instruction which matches a "call" instruction which also had the least significant bit as a 1. As such the program counter is simply set equal to the value of the old program counter.

If A has the value 0, then a more complex return is needed. The algorithm is the reverse of the call instruction. Therefore the function to handle the "return" instruction should be able to differentiate between the two different calls.

3.6.12. Switch State

There are two states in the machine which indicate which of the two register sets is the current set. The execution of the Switch State instruction changes the current set to the other.

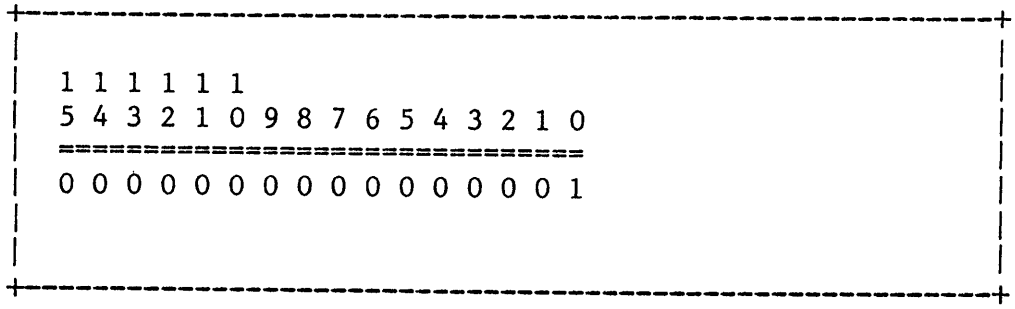


Figure 3.25 Switch State instruction

If the kernel wishes to dispatch a process it simply loads the other register set and then executes this statement.

When a non-kernel process wishes to have the kernel perform some operations it simply executes this instruction and the kernel will continue from the location after the one at which it executed the switch state instruction that dispatched the process.

When this instruction is executed, certain bits in the status register get set to indicate the reason. Trace trap testing will be disabled until after the next instruction.

The Switch State instruction definition implies that it could be implemented in two different ways.

1. Two vectors could have been used to hold the values of the two register sets and access the values through a pointer.

2. One vector could have been used to hold the values of one register set and copy the values of the register set in use.

3.6.13. Implementation Details

The code for the ALU process was written in different stages. The stages were as follows :

(i) Wrote a function to initialise all variables called "Initialise".

(ii) Wrote a function called "Run" to select the instruction to be executed and then call a function to execute the instruction.

(iii) Wrote each of the functions which execute an instruction.

(iv) Wrote all the support functions to do minor tasks.

The "Initialise" function is very small containing all the necessary initialisation statements. This is the function that receives the messages from the program "Set_machine" and sets up the communication path for the ALU. It also initialises the program counter register and the frame pointer registers to their initial values.

The "Run" function has the main control loop. It repeatedly reads instructions, select the most significant non-zero bit and calls the appropriate function to execute that particular instruction.

Originally instructions were executed silently without the use of the screen. After the Front panel process was created the ALU could send messages to the front panel to display certain information. More will be discussed about the Front panel later. The Run function sends a SERVER_READY message to the Front_panel

when it is ready to execute instructions. After sending a SERVER_READY message it calls the function "Get_next_inst" which gets one instruction from memory. It then calls the function "Inst_length" which returns the number of instruction packets needed to execute this particular instruction. If more instruction packets are needed, either another 1 or 2 are obtained.

After the necessary packets are obtained, and if the instruction has to be displayed on the screen, the function "One_instruction" is called to take either 1, 2 or 3 instruction packets and concatenate them into one string. After the instruction string has been formatted "Update_inst" is called to update the instruction display on the screen.

The main purpose of the "Get_next_inst" function is to obtain the next instruction from code memory. The original design of this function was very inefficient because, every time an instruction was required it had to send a request to memory which can be a very slow process. To improve the speed an instruction cache was introduced. At this point in time the simulator was driving the design of the machine. By observing the execution of the simulation, improvements to the machine design were made possible. One such improvement was incorporating cache memory.

3.6.14. Cache (Memory Prefetch)

The main purpose of having on board cache memory is to reduce bus contentions. Having a local area of cache for each

processor is very effective particularly in the case of a program executing a loop which has all the instructions within the cache area. The maximum capacity of the cache is 256 bits. This is implemented as a linked list of four nodes based on the following template.

```
LINK     : &Cache_entry
ADDR     : unsigned[32]
VALS[4]  : unsigned[16]
```

The field ADDR contains an instruction or data address, the vector VALS contains the instructions or data that are at ADDR, ADDR+1, ADDR+2 and ADDR+3. When the function "Get_next_inst" is given an address, it checks the linked list to confirm if that particular address is in the linked list. If the address is found to be in the linked list, it then moves that particular node to the head of the list, and returns the values in vector VALS. As can be seen above, there are four instruction packets in one node, therefore the probability of finding the next packet in the same node is very high. This is the main reason for choosing this data structure as opposed to an array. Using an array structure would be too slow because every time a value is required, the search has to start from the beginning of the array or a wrap around index would have to be used. To avoid such a situation the array would have to be rearranged each time a value is found, which would still be uneconomical.

If the value is not found in the linked list, then a request is sent to memory to obtain another 4 packets of information. The new information is put at the head of the linked list. Each time memory is accessed, 4 packets are obtained because the memory is read 64 bits at a time.

Caching was also done for data memory access. The only data that cannot be cached is the Input/Output(I/O) data. The "Get_next_data" function makes certain that the data required is not I/O data before it checks in the cache. At the initial stage, the data cache did not exist, but adding the code necessary to implement a data cache was very simple. The only function that had to be changed was the function "Get_next_data". This confirms the advantage of using functions and processes to do different tasks. The only process affected was the ALU, because the function "Get_next_data" resides as part of the ALU process. This also takes advantage of the PORT scope rules since only the function changed could be affected.

When the "Get_next_data" function was first changed to introduce a data cache, there was a bug which was not obvious. The linked list used to store the data cache was loaded with the same value twice. Therefore the program worked but the loading of the data cache was not executed correctly. When the addresses of the data cache were displayed on the screen using the Front Panel process the bug was detected. This was a good example where the simulator was used to help in debugging the simulator itself.

The function "Assign_history_regs" was written to modularise the code because each time the program executes a Jump, Call, or a Flying_leap, the history registers have to be changed.

3.6.15. Sending Messages to the Screen Process

A function called "Send_to_window" was written which has the following arguments.

```
which   : unsigned {field number on the screen}
address : unsigned[32] {contents to put on the field}
request : unsigned {how to display the contents}
```

When the function receives the above arguments from the calling process it assigns the values to the appropriate fields in a message and sends the message to the Front Panel process. This function is called by many functions which require display information on the screen. The function was created to reduce the code size, so that all the functions that have to send information to the screen do not have to set up a message and send the information. The pseudo-code version of the function follows.

```
WHICH[message]    = field number
ADDRESS[message]  = field content
REQUEST[message]  = request
send message to Front Panel
return
```

The same function discussed above was used by various processes, such as the MMU process. This method would also make the calls to display information by different processes consistent. By making the function consistent, making changes to its content is much simpler.

Whenever register contents have to be updated the function "Update_registers" is invoked. This function uses a variable called "Base" which is the screen field number of register 0. When the other registers have to be accessed their manifest constant is added to the Base field to define their screen field number. The contents of the registers are accessed by their manifest constants. The contents of the registers are in an array which makes it uncomplicated to refer to a particular register by simply indexing the array by the manifest constant. Any other data structure would have been very time consuming to code and to implement. The function "Update_registers" invokes "Send_to_window" each time a register value has to be displayed on the screen. Registers other than the program counter register get updated on the screen only when tracing has been turned on.

The function "Redraw_prefetch" is used to display the addresses that reside in the cache. This function is called each time an address has changed within the linked list. Similarly the display is updated only when tracing is required.

3.6.16. Monitoring the Execution of ALU

The monitoring is done to find out the efficiency of different program modules. This is important in order to detect which parts of the programs are being executed inefficiently. The word "efficiency" in this context means the measure of the number of instructions taken to execute a certain task. Efficiency increases as the number of instructions executed decreases. To measure the efficiency, a general approach which is similar to the tracing of the MMU has been used, therefore this discussion will be on the modifications that were made to the program code and not how the tracing was done.

Code size is very important when writing programs, but another equally important aspect is the number of actual machine instructions that get executed to do a certain task. It is possible to write a program which is rather small in code size, but the number of machine instructions it executes is far too large. The best way to start to reduce the number of instructions that are executed is to run a sample program and do some experiments.

A test program was written using the new machine's instruction set to work out the dates of Easter until the year 2099. The tracing was done whilst working out Easter for one year. At the initial stage the ALU of the simulator used approximately 2958 instructions for each instruction packet simulated. When the trace by function information was obtained, it was obvious which functions were executing an excessive amount of instructions. The function "Add" used close to 200,000 instructions.

All the other major usage was in the functions that were used by the function "One_instruction".

The first attempt to reduce the number of instructions was concentrated on the functions that were used by "One_instruction". This was done because the function "One_instruction" was not part of the actual hardware design of the ALU. Another reason was that the function "One_instruction" was invoked many more times than the function "Add". The function "One_instruction" was invoked for every instruction that was executed. Looking at the code, it was clear that most of the functions were doing things that did not need to be done or that they were written very inefficiently.

After modifying the functions that were inefficient, the next version used approximately 1447 instructions for each instruction packet. In the first version, the function "One_instruction" was called each time an instruction packet was obtained. After the modification, the "Run" function reads an instruction and used a function called "Inst_length" to work out how many more instruction packets it needs to execute. If more instructions are needed, the next 1 or 2 instructions are read and then the function "One_instruction" is called to make up the string which contains all the instruction details to be displayed on the screen. The function "Inst_length" uses very few instructions, comparatively the function "One_instruction" uses many more instructions. Therefore adding a new instruction and reduc-

ing the calls to "One_instruction" reduced the number of instructions executed.

Looking at the function "Add" it was seen that the algorithm used to add two binary numbers was very inefficient. The algorithm needed too many 32 bit shifts. Doing 32 bit shifts costs too much in the number of instructions that are needed to execute the operation. After improving the algorithm for "Add" and making more modifications to the function "One_instruction" and its member functions, the ratio of machine instructions vs. instruction packets were reduced down to approximately 1074. Compared to the original code, the saving on instructions is approximately 60%.

The evidence emphasises the importance of tracing code not only for debugging purposes, but also for writing better code and saving computing time. There is a very high probability that most programs can be made more efficient after the first attempt at coding a program. After the code for the ALU was made more efficient, the execution speed was noticeably higher.

3.7. Memory

The memory process simulates the storage and retrieval of data and code in the machine. The machine uses different sections of memory to store related contents. For example, the data values are stored within a certain address range, and the code is stored in a different address range. Memory processes receive messages from the bus requesting certain actions to be performed,

such as reading or writing data. The replies are passed back to the bus via a gofer. The memory is simulated using files to store information.

3.7.1. Obtaining values from memory

The contents of simulated memory could have been stored in memory or on disk. Storing simulated memory in real memory is impossible because of the size of the physical memory available. Therefore contents of simulated memory were stored on disk. To reduce the disk accesses needed to obtain values from memory a blocking mechanism was used. Each time the disk was accessed a block of 2048 bytes was obtained. This method has a distinct advantage over a non blocking system because the memory process has to access the disk fewer times. Disk accesses can be time consuming, therefore each time the disk is accessed, a block of 2048 bytes is obtained. Each time memory is read 64 bits of information from the current block are passed back to the calling function.

As caching of memory saves on requests to the memory process and bus contentions, the blocking mechanism implemented in the memory process saves on disk accesses. In most machines the input and output are the most time consuming, therefore any saving on time made in this area is a major advantage. The simulator was used to experiment and work out the optimum block size for reading information from the disk.

3.7.2. Implementation details

The program "Set_machine" sends messages to the process "Memory" with all the information it needs to set up its own communication path. The "Memory" process has to receive these messages and reply back to the program "Set_machine" indicating it received the information successfully. The initialisation statements are as follows.

```
receive (Message)
low      = ADDRESS[Message]
high     = DATA[Message]
receive (Message)
in_id    = SRC_ID[Message]
receive (Message)
out_id   = SRC_ID[Message]
```

The variable "in_id" is the process number from which the "Memory" process will receive messages. The "low" and "high" are the address range, "out_id" is the process number to which the "Memory" will send messages.

After the initialisation has been done, the particular memory process knows its communication parameters. The process "Memory" has a main loop to repeatedly receive messages from its "in_id" and executes the instruction. This loop has the following structure.

```
repeat
        receive (Msg, in_id)
        process the request
        send (Msg, out_id)
end;
```

Memory contents are stored in PORT files on the hard disk as a sequence of bytes. The contents of the block of memory are stored in an array data structure. An array data structure has been used for the purpose of simplicity. Indexing into the array structure is very simple. The following statement works out the index.

```
index = address & $7FF
```

A bitwise "and (&)" of the address field by $7FF gives an index between 0 - 2048. The value 2048 is not a magic number, rather it is an optimum number chosen after testing and timing the disk accesses.

The PORT file system allows reading or writing to be done on arbitrary sized blocks. The procedure "Load_block" loads a block of data into the variable "The_block" which is the variable which contains the current block of memory. When the function "Load_block" is called, it checks if the required block is the current block in memory. If the memory required is not in the current block, then a new block is read in to the variable "The_block". Before the new block is read in, it has to check if

the current block is dirty. If the block is dirty, then it has to be written out to disk.

3.8. Bus

This is the process which carries data to and from memory. It sits between one simulated board and the global communication path. The code for this process provides the interface logic of the simulated board.

Each memory process has a bus process which belongs to its address space. The purpose of the bus is to trap addresses that belong to its board and send the messages to a gofer which will pass it on to the memory process. The bus processes are connected in a loop, i.e. the design of the communication path is circular. If an invalid address is sent from the MMU to the communication path, then none of the busses will recognise the address, therefore the address will go around the path and come back to the original bus. When an address comes back to the bus which initiated it then that particular address is invalid. This is an asynchronous access method where a process waits until it receives an acknowledgement from the servicing process. The above method of detecting errors is useful in eliminating controls such as a minimum waiting time for a service to take place if a synchronous method was used. This also allows the freedom to order "boards" arbitrarily.

3.8.1. Initialisation of the bus process

The bus process initialises itself by receiving messages from the program "Set_machine". Even though there is only one code file to simulate the bus, when the machine starts up more than one bus process is required. The "Size" field mentioned in 2.4.1.3 are very important for the bus processes because the sizes define which address space that particular bus accepts.

Changing the address range of a bus process is a very simple task which only entails changing the address range in the text file which is used by "Set_machine" function. This is another advantage of using the text file to configure the simulator. Not only is changing the parameters of the existing bus processes simple, but adding a new bus process is as easy as adding another line of text in the file.

3.8.2. Outgoing message

The bus processes send messages to two different processes. They are the next bus process and its board(gofer). The Out_ids in the text file specify to which processes the bus can send messages. There are two types of outgoing messages. One is when the bus sends a request to the next bus because the address did not belong to its board. The second type is when the bus requests its board(gofer) to process the message and take some action because the address was for its own board.

3.8.3. Incoming messages

The bus process receives messages from two processes. The two processes are the previous bus and its board(gofer). The In_ids in the text file specifies from which processes the bus can receive messages. The following are the types of incoming messages.

From Gofer to the Bus

1. A request to perform some actions.
2. A reply to indicate it processed a message.

From Bus(n-1) to Bus(n)

1. Request to perform some actions.
   The actions are either process the message or pass it on to the next bus.
2. An invalid message, which has cycled around the communication path.

When a bus receives a message from the previous bus, it will process the message if the address belongs to its board. If the message belongs to its board, the bus has to request its board to process the message. After the bus has sent the request to process the message and the action requested has been completed the bus will receive a request from its board(gofer). After the bus receives the request from its board, it will send the message along the communication path.

If the address did not belong to its board, then the bus will request the next bus to process the message. If the

original bus which sent the address receives the same address from a previous bus, then that address is invalid.

3.9. Front Panel Process

This process has to perform two major activities :

1. Display simulated information on the screen
2. Handle user input (i.e special keys and alphanumeric characters)

The front panel process receives from all other processes in the simulator, but it is the only process which does not send to other processes created by the simulator. Therefore initialising the front panel process involves receiving all the In_ids from the program "Set_machine". A vector of local In_ids was created to store all the In_ids rather than use different names to store all the data. The vector allows future changes to the In_id list without having to change the code. Storing the Ids will be discussed later.

3.9.1. Implementation of the Front Panel

The screen is designed using a program called "fg" provided by the PORT system. This allows the programmer to designate all screen details. The details are as follows:

Field Location (i.e. X & Y coordinates)

Field Length

Field Type (i.e. String or Numeric)

The program "fg" uses the above information to create the following external variables:

Field_numbers

Field_rows

Field_columns

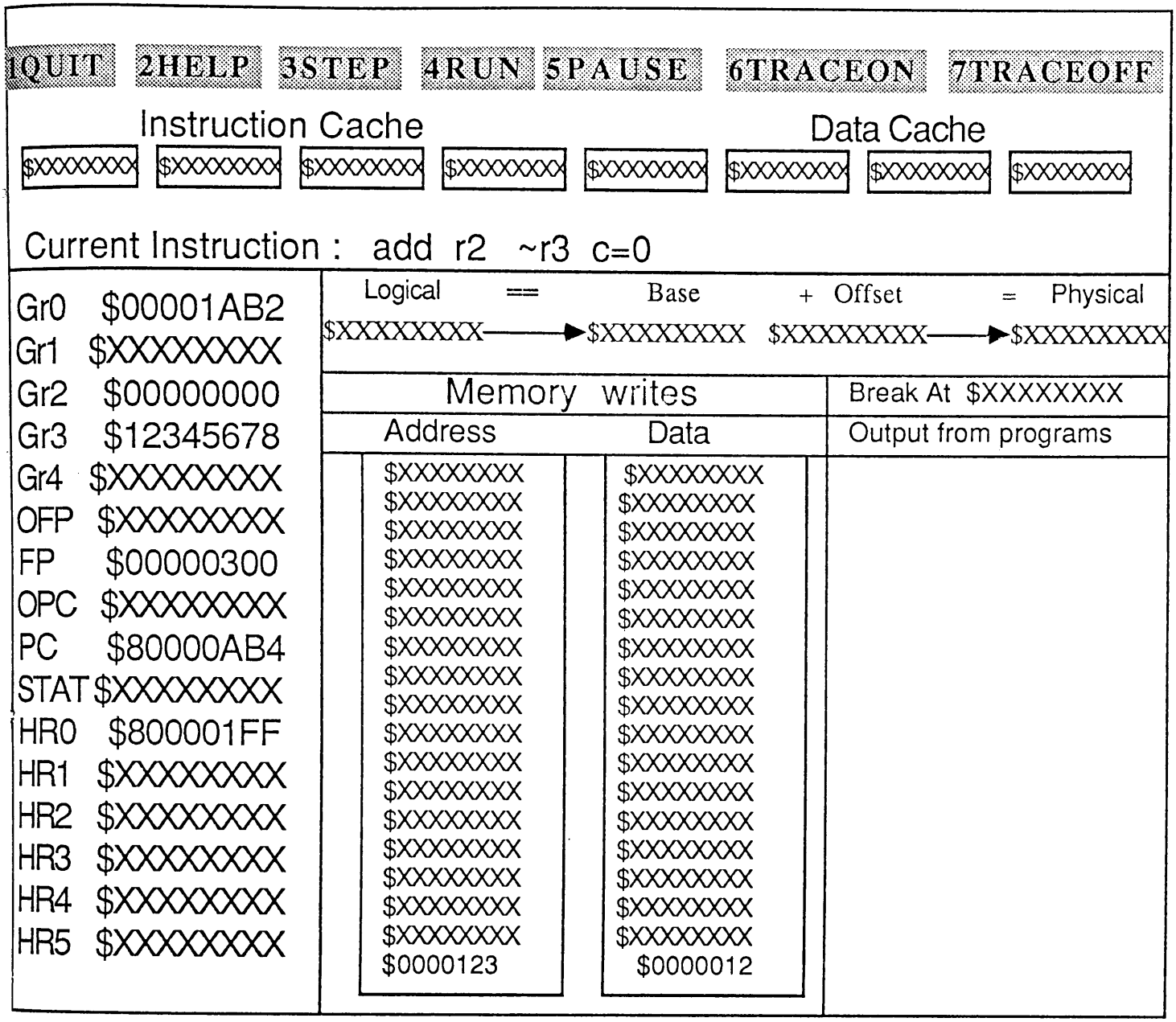
Field_types

Field_widths

Screen_image

All the above variables are self explanatory, except for "Screen_image" which is a copy of the screen image as designed by the programmer. This image can be edited and changed by the programmer. The screen was designed in two stages, at the initial stage an area was reserved for future use.

Diagram of a screen format follows



GR - General Register
FP - Frame Pointer
OFP - Old Frame Pointer
PC - Program Counter
OPC - Old Program Counter
STAT - Status Register
HR - History register

PORT also provides most of the screen initialisation and startup functions. The Window Manager of PORT provides part of the building blocks necessary to implement a user driven screen. The main functions that had to be written and modified were as follows:

Update_field

Scroll_memory_writes

Update_inst

Server_ready

Input_arrived

The function "Update_field" is called when the contents of a particular field on the screen have to be changed. The function has to be given the field number to be changed and the data to be displayed. From the field number, the row and column can be obtained by indexing into the appropriate vectors created by the program "fg".

"Scroll_memory_writes" function is used to display the addresses and contents of memory. When the area allocated for memory is full, the contents on the screen get scrolled up to make room for the new values. Except for the scrolling mechanism, this function works in a similar manner as the "Update_field" function.

The function "Update_inst" displays the next instruction to be executed. The instruction is displayed as a string. The string to be displayed was concatenated in the function "One_instruction" as mentioned in the discussion of ALU. The string contains the machine code value and the assembler equivalent. The following is an example of the display:

```
$8024      add  r2  r4
```

The "$8024" is the machine instruction representing the addition of register "r2" with register "r4" and the result being placed in "r2".

The function "Server_ready" is called when the ALU sends the "SERVER_READY" request. When the ALU is ready to execute an instruction it sends a "SERVER_READY" request to the front panel. When this request is received, the front panel sets the machine to the appropriate state and redraws the active buttons. If the current state permits the execution of another instruction, then the front panel will give permission to the ALU to execute the next instruction.

The function "Input_arrived" handles all keyboard input. The keyboard input consists of alphanumeric keys and special keys, all control keys are ignored. When any key on the keyboard is pressed, this function will interpret the input key and call the appropriate function to handle that particular key.

3.9.2. Screen control keys

To a certain extent the output on the screen can be controlled by the user. The IBM-XT keyboard contains 10 special keys (pf1 - pf10) on the left hand side of the keyboard. These keys allow the user to select options on the screen when the simulator is in motion. For this simulator only 7 out of the 10 options are used. The need for alphanumeric characters will be discussed in the section about break points. The ALU is controlled by the input from the user. Since the user input is handled by the Front panel the execution of the ALU is controlled by the Front panel. The following is a pseudo code version of the ALU control.

```
while there are more instructions to execute
        ask front panel for permission to execute
        execute the next instruction
        update the screen
```

This allows the user to control the execution of the machine.

3.9.2.1. Special Keys

Only certain keys shown in the diagram are recognised by the program when pressed by a user at any given time. All other key presses are ignored. The keys that are recognised by the program are highlighted. As can be seen from the diagram, the "quit" option is the pf1 key. This key is pressed to stop the execution

of the simulator. When the execution is stopped by pressing pf1, all the processes get destroyed(killed). Destroying the processes is important because the processes may not terminate unless specifically destroyed or the system is rebooted. The processes could have been written so that each process destroys itself when the processes it communicates were non existent. This method assumes that the processes are communicating as expected. If there was a bug in one of the processes however it may not destroy itself, thereby affecting all other processes dependent on that particular process.

The pf2 key is the "help" key. The operation of the simulator is self explanatory. A help option still exists to provide information about the simulator to users. When the pf2 button is pressed, the program looks for a text file called "help" and displays the contents of the file. Allowing the user to page back and forth.

The "Step" option (pf3) key is used to execute the simulator one instruction at a time. Since the ALU has to ask permission from the Front panel before it executes an instruction, only when this button is pressed can the ALU execute the next instruction. The "step" mode is very useful when the user wants to look at the changes that took place after a single instruction was executed. This mode allows the user to observe the changes at his/her own pace. It also gives the user enough time to write down notes if necessary.

The "Run" (pf4) key is the automatic mode button. The pf3 is the manual (step) mode, so when the pf4 key is pressed, the Front panel gives the ALU permission to execute the next instruction every time. It stops giving permission only when the pf5 button is pressed. This button is necessary because a user may want to speed up the execution or go away and leave the machine running. This button essentially simulates an user rapidly pressing the "pf3" button.

The "Pause" button (pf5) is absolutely necessary because a user may want to stop the execution of the machine at a given time. This button will be necessary only when the machine is in the "Run" mode which was discussed in the previous paragraph. When the "Pause" button is pressed, the Front panel does not give permission to the ALU to execute an instruction. Therefore the ALU waits until it is allowed to execute an instruction.

The "Traceon" (pf6) key turns the tracing mechanism on, and the "Traceoff" (pf7) key turns the tracing mechanism off. The trace mechanism discussed here is mainly concerned with running a program in the machine and observing the execution. When the trace is on, all changes to the machine are displayed on the screen. When the trace is off only the program counter changes are displayed on the screen. When the trace is off, the machine executes at a much faster speed. This is mainly because most of the time is spent on making changes to the screen or creating readable instructions. The program counter is displayed so that the user knows that instructions are being executed.

3.9.2.2. Implementation of the special keys

There are seven buttons in total which the user can use to control the screen displays. Writing the code to handle the user control inputs can get very complex. When one or more of the buttons are pressed, the machine is in a particular state, so when another button is pressed, the state of the machine has to be considered before any action can be taken on the latest input. Before writing any code, a state table was built to work out which buttons are allowed to be pressed at a given point and what action is to be taken when a particular button is pressed. A two dimensional table with buttons as columns and states as rows was created. After designing the transition table on paper, all the conditions were incorporated into the two dimensional table called "Trans_table". The following are a few examples from the "Trans_table".

```
Trans_table[0][2] = 1;
Trans_table[1][3] = 2 + $20;
Trans_table[1][7] = 4;
Trans_table[5][2] = 5 + $10;
```

A variable called "State" was used to indicate the current state of the machine. When a new "pf" key is pressed, the "Trans_table" is indexed by the value of "State" and the key number. The value obtained contains the new State of the machine and an indication whether to reply to the ALU. The State of the

machine is in the least significant 4 bits of the value. The reply details are in the upper 12 bits of the value.

Another one dimensional table called "Soft_buttons" was created. The table indicates which buttons can be pressed by the user at a given state of the machine. This table is indexed by the variable "State" which returns a value which indicates which buttons are active (i.e. which buttons cannot be pressed). The following are a few example rows from the table "Soft_buttons"

```
Soft_buttons[0] = PF3_ACTIVE | PF4_ACTIVE | PF5_ACTIVE |
                  PF6_ACTIVE | PF7_ACTIVE;

Soft_buttons[3] = PF3_ACTIVE | PF5_ACTIVE | PF6_ACTIVE;

Soft_buttons[8] = PF3_ACTIVE | PF4_ACTIVE | PF5_ACTIVE |
                  PF6_ACTIVE | PF7_ACTIVE;
```

When any of the special keys are pressed, the function called "Input_arrived" is called. This function decides which key was pressed and calls the appropriate function. Each key is handled by a different function. Each of the functions call the function "Change_state" and pass their key number (i.e. the column number to index into "Trans_table"). The function "Change_state" will change the variable "State" to its new value. The following is an algorithm used by the function "Change_state" to change the state of the machine and reply to ALU if necessary.

```
value = Trans_table[State][key]
State = value & $F        {get the new state of the machine}
 {decide which buttons are active in the current state}
The_active_buttons = Soft_buttons[State]
value = value >> 4   {shift down the lower 4 bits}
if (value != 0)      {check if the ALU should be replied to}
    reply to the ALU
```

The alternative way of implementing the code to handle the special key inputs would be to have a string of conditional checks. This method would entail the code having to check what the previous state was, and then changing the current state. After that it has to do a conditional check to work out which buttons are active in the new state. The code required to implement this type of algorithm would be too long and difficult to maintain. In the current implementation, getting the new state requires only one statement, and working out the active buttons requires only one statement. This method is also faster because the external tables "Trans_table" and "Soft_buttons" are set up at compile time and can be simply indexed at execution time.

3.9.2.3. Break Points

The Front panel's function "Input_arrived" handles alphanumeric input as well as special key input from the keyboard. The keyboard input is necessary to incorporate break points. When the simulator is executing in the "Run" mode, the user may wish to stop the execution when the program counter (PC) is equal to a certain value. To do this the user could watch the value of the PC and press the Pause button when the PC value is

equal to a pre determined value. As can be seen this would be impossible to do because of the speed at which the value of the PC changes are displayed on the screen. Therefore the concept of break point was introduced. The user is allowed to type in a break point value, when the PC is equal to this value the Front panel pretends the user pressed the pause button at just the right time. Since the ALU cannot execute without the permission from the Front panel, the machine will pause.

The break points are a vital part of the simulator. The break point mechanism allows the user a simple and elegant means of debugging programs. Any debugging tool would not be complete if an user was not allowed to set break points.

3.10. On Board Switch (OBS)

As the name suggests, this process only switches the direction of an address. When an address is sent to the OBS by the MMU, its only task is to trap a set of addresses and send it to one of two pieces of hardware depending on certain conditions. The most common use of the OBS would be for on board memory. The following pseudo-code explains the task of the OBS.

```
if (Address >= low && Address <= high)
        send Address to first piece of hardware
else
        send Address to second piece of hardware
```

In the initial design of the machine, this process was not part of the machine. This process came into existence only when there was a need to display characters on the screen area which was kept for future use. The OBS process has a very simple function to perform. In the current implementation, the OBS sits between the MMU, the Serial Line and the Gofer which sends messages to the first bus.

Adding this process to the simulator was a simple task. Since each process can be coded separately the other processes in the simulator did not have to be modified. This highlights the advantages of a multiple-process simulator. In the initial design, the MMU would send a message directly to the gofer which in turn would send the message to the bus. After the OBS was introduced the code in the MMU still appears to be sending messages to a gofer, but the text file was changed so that the MMU sent messages to the OBS. This was achieved without having to change the code for the MMU or the gofer.

3.11. Serial Line

Part of the screen was reserved for future use and this is where that area will be discussed. The Serial Line is a cheap peripheral attached to the machine which provides simple output for the machine. The Serial Line process receives input from the OBS and sends the input received to the Front panel. On receiving input from the Serial line the Front panel will display the input in the area reserved for display purposes. Displaying out-

put produced by programs running in the machine was very useful when testing the machine's execution.

3.11.1. Implementation

The implementation for the Serial Line was very simple because the only job of the serial line is as follows:

```
repeat
  receive a message from OBS
  reply to OBS
  send the data to the front panel to display
```

A function called "Serial_line" was created, this function receives the In_ids and the Out_ids from the program "Set_machine". After the initialisation has been done, the function repeatedly receives messages from the OBS. The contents of the addresses are then passed on to the Front panel to be displayed on the screen. This process was tested when testing the whole simulator. To test the serial line, test programs were written to display messages on the screen. It is also worth noting that the serial line was introduced to the machine only when there was a need to display output from the test programs. This peripheral was introduced without having to alter any of the other functions.

4. Testing The Simulator

To test the simulator to a reasonable extent, many programs were needed because it was decided to test the simulator with some real data (i.e to simulate a program that did something sensible). The programs that were used to test the simulator will be mentioned after discussing the creation of such programs. To run the simulator, the contents of code and data memory had to be created. A decision was made to create two files to simulate code and data memory. The code file had to contain machine instructions (in binary). The data file had to contain data in binary. Creating the code file was going to be the major hurdle. There were two possible ways of creating the code file, they were:

i) Write a program in machine code by editing a file.

ii) Write a program in assembler and assemble the program to create a file in machine code.

At first, it may seem that the first option is easier because an assembler had to be written to implement the second option. Looking at future needs the second option was the most sensible. If the first option was chosen, then creating and checking the code would be a tedious task. This task would have to be repeated many times.

4.1. Writing An Assembler

A simple assembling program was written to parse instructions written in assembler. This program creates both the code and data files, it also produces an assembler listing for reference. The intention was to produce the necessary files, therefore in some cases efficiency and elegance were sacrificed for speed of production.

4.1.1. Implementation details

The implementation of the assembler was kept as simple as possible. The main loop reads a line of characters and calls an appropriate function depending on the first word of the line. The first word is an instruction name except for a few words. When the word ".code" is encountered, the code file is chosen as the output file, when the word ".data" is encountered the data file is chosen as the output file. The words such as "byte", "half-word", and "word" are used to set up data values of those sizes. The word "Label" is also recognised which defines the position of a label.

When a word matching a defined instruction is encountered, a function which performs the necessary conversion to machine language is called. All addressing mode values are worked out where necessary and written to the appropriate place. If a reference to a label is made, the parser will check a linked list of labels to determine the address of the label. If the label is found, then its address is written adjacent to the instruction.

When forward references are made on instructions such as "call", "jump" or "if", then the linked list of label definitions will not contain the label. At this moment the parser writes zeros in place of the label address and stores the byte position of the code file, line number and address of the instruction in a linked list.

At the end of the first pass, the program consults the linked list with the undefined labels and takes the addresses from the linked list with the defined labels and writes the addresses to the code file at the correct byte location. If an undefined label is not found in the linked list with defined labels, then an error condition is returned.

An assembler listing is also produced at the end of the compilation. This listing contains the program counter, instructions in machine code and the assembler version of instructions.

4.1.2. Using the Assembler to create test programs

The assembler program was used to create code and data files which calculate the dates of Easter for 99 years. This program used sub routines such as "divide" and "multiply" to test arithmetic and call instructions. The program utilised every piece of hardware to perform the required calculations, storage and display. The date of Easter for every year was displayed on the screen using the "serial line" process. This was the main program that was used to test all the simulated pieces of hardware

and instructions, other minor programs were used when testing individual pieces of simulated hardware.

4.2. Writing a disassembler

After the assembler was written to produce the code file, it was rather strenuous to check if all the instructions were correct. The only way to check the instructions was to check each one by hand, or to write a disassembler to produce the original assembler instructions. The second option was more feasible because of the large number of instructions that had to be checked.

4.2.1. Implementation Details

An elementary but efficient disassembler program was created to check the data. This took little time to write and after it was written it was simple to check the data in the code file.

The disassembler program reads the instructions from the code file and produces a assembler listing. This program reverses the action taken by the assembler program. The highest bit that is turned on marks the type of instruction. The other bits are used in certain instructions to decide if more bytes have to be read before reading another instruction. This program has a simple loop which reads an instruction from the code file and calls an appropriate function to handle that instruction. The concept of having a different function to handle each instruction makes it very comfortable to maintain the program. This method

makes debugging a very simple task because it isolates the problem area to a particular function.

4.3. Debugging the two programs

As discussed above, the assembler and the disassembler programs were built separately. Each program has a unique task to perform, the two tasks are the reverse of each other. Since the assembler and the disassembler were both written as separate entities, the possibility of making the same coding error in both programs is very remote. Both programs were coded from matching specifications, therefore any coding error occurring in one program will be highlighted by the other program.

Consider the following example:

```
Load  r0  r2 [2]
```

When the assembler program parses the above instruction it will decode the instruction into the mode, registers and offset. The disassembler program uses the decoded bits to obtain the original instruction. Therefore if the assembler had decoded the instruction incorrectly, then the disassembler will not be able to produce the original instruction correctly. If the disassembler program produced a wrong instruction, this would throw light on the fact that one of the programs was wrong.

5. How to use the simulator

The first objective is to create a file with machine code and a file with data. This can be achieved by creating a file in assembler (refer to Appendix II for mnemonics). After the file has been created, it can be assembled by the assembler program. The simulator program looks for the code and data files in the temporary directory in the file tree. The code file has to be named "_..............................." and the data file is named "...............................". This unusual naming convention has been chosen to allow different files to be used to simulate different address ranges in memory. Also to simplify name generation and not conflict with any useful names. The file name represents the 32 bits in an address. Every "_" represents a 1 and the "." represents a 0 value. Consider the following example:

Address = $00040000

The file name representing this address would be :

".............._................."

The files that are created by the assembler are called "Code" and "Data". Therefore these files have to be copied to the above mentioned names in the correct directory before running the simulator.

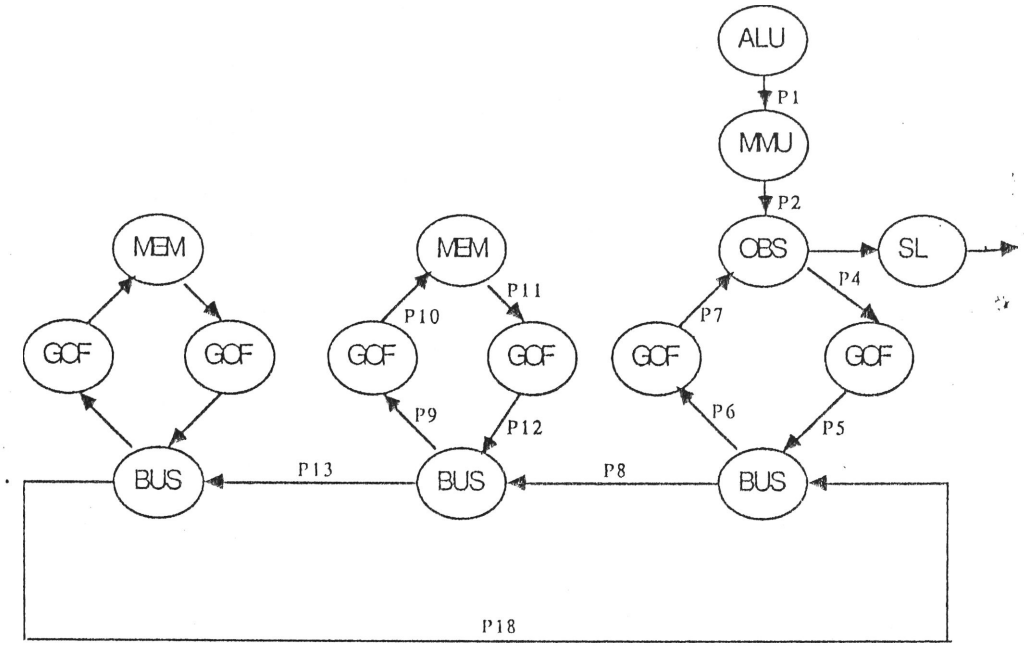
After the code and data files have been created, running the simulator is simple. To start the simulator, simply execute the program "Set_machine". This program will start the simulator, create all the necessary processes and set up the communication path. After the simulator has started to execute, the user could control the execution with the special keys. These keys give the user full control of the simulator's output on the screen. The output provided by the simulator is self explanatory, so the details will not be discussed.

5.1. How the simulator works

Details of all the simulated hardware components have been discussed in chapter 3. The following paragraph will discuss their usage within the whole machine. The discussion will be based on specific examples so as to facilitate the understanding of the simulator's execution.

5.2. A sample simulation of a program's execution

The example that will be discueed in detail is as follows: Please refer to figure 5.1.



P1 - P18 Represent the Communication Path

Figure 5.1 Communication Path

The instructions being simulated take a string of data from memory, add a value to each character and then display each character. The discussion will be based on the communication that is required to perform such an operation.

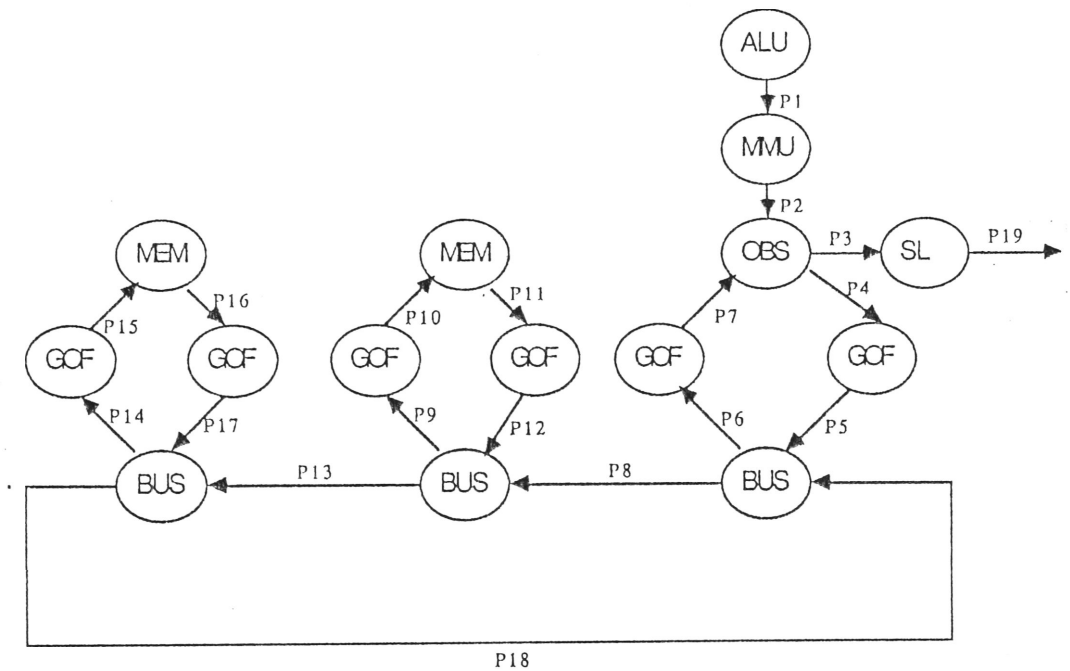
The program is created using the assembler program. Instructions are typed in using the editor in a format the assembler expects (refer to Appendix II). This file is converted to machine instructions by the assembler and then written to the file which contains instruction packets. All data values are written to another file which stores data packets. Code addresses start at $80000000 and data addresses start at $0.

When the simulator first starts to execute, the assumption is that all registers are initialised. ALU sends a request via P1 to the MMU seeking the instruction packet at location $80000000 (code addresses start at this value). MMU does some address translation and sends it along P2. The OBS will check the address and decide to send it along P4. The Gofer will then pass it along P5. The bus will then pass the address along P8. If the address does not belong to that particular bus, it then sends the address along P13. If the address is valid, then the bus which belongs to that address range will respond and accept the address. After the address has been accepted, the bus will send the address along P8. The gofer will then pass the address along P10. After the memory receives the address, it converts the address to a location on the file and reads a block of data (code file is used for instructions, data file for data values). It then sends 64 bits of information to the bus via P11 and P12. The bus that received the contents of memory will send it along to the next bus (p13) which in turn will send it to next bus to be passed along p6. Then the values will be passed along P7 to the gofer which passes it along, P2 to MMU, which then will pass the contents via P1 back to the ALU.

After the ALU receives the machine instruction, it stores the information in the instruction cache. If all the details of this particular instruction have been obtained, the string representation of the instruction is sent to the front panel to be displayed. The instruction is then decoded by the "Run" function and the function "Load" will be invoked to execute that

instruction. To load a character from memory, the ALU has to go through the same sequence of tasks as mentioned above. The only difference is that this time the data memory will respond and pass back 64 bits of data. This data will be then stored in the data cache.

After the character has been loaded, when the next instruction is needed, the ALU will first look in the instruction cache which will contain an "add" instruction. As can be seen, obtaining the next instruction was very much quicker and easier. After the "add" instruction has been obtained and decoded the function "Add" will be invoked to add the contents of two registers.



P1 - P18 Represent the Communication Path

Figure 5.2 Communication Path

After the add instruction has taken place, the new value has to be stored. ALU then sends a request via P1, on receiving the

request the MMU then sends the address along P2. If the character is to be displayed on the screen, then it is sent along P3. On receiving the character the Serial Line will send it to the Front Panel to be displayed.

If the character does not have to be displayed, then the contents and the address reach the memory process via the same path mentioned above. Memory then writes the contents at the given location and replies back via P16. When a value is written to an address in the current block, that block is said to be "dirty". If a block is "dirty", it has to be written back to the file before another block of information is obtained from the file.

The above process continues until the whole string has been processed. If the loop needs less than 16 instruction packets, then all the needed instructions will fit into the cache, thereby eliminating the need to access memory. Another point to note is the contents of registers; each time the register values change, the new values are displayed on the screen if the trace has been turned on.

5.3. Concluding Remarks

The simulator has been very helpful in developing this new personal computer. It has made it possible to highlight many design errors and better methods of designing the hardware components in the machine. For example in the initial design of the machine cache memory was not part of the design. But after test-

ing some programs using the simulator it was seen that cache memory would speed the execution of programs. This simulator can be further utilised to help in designing an Operating System and/or a compiler. All further development work on the machine will be done using the simulator as a guide to visually observe the execution of the machine instructions.

If a compiler were to be designed, the simulator can be used quite effectively to help in debugging the code. The output produced by the simulator has been designed to give maximum help in debugging programs. The disassembler program can also be used to debug code images produced by a compiler. The development work on the simulator has been exciting and thought provoking. There is a lot more work left to be done to complete the design of the whole machine. Now that the foundation has been laid further development can be done more easily with the aid of the simulator. This project was the first attempt at the design of the machine. In the light of these circumstances, writing the simulator was a time consuming task. The method that was chosen to implement the simulator was the best under those circumstances. The simulator is so flexible that making changes to each component of simulated hardware was rather simple because each piece of hardware was simulated by a different process. As can be seen, if any other method was used to simulate the hardware components, making changes to the design would be rather laborious.

APPENDIX I

I.1 Code for ALU "Run" function

```
    import( IO_request,
            IO_requests,
            Registers )
()

    high_bit : unsigned
    request  : IO_request
    num      : unsigned

  {
    Pre_base = 18;
    repeat
      {
        Send_to_window( Base_field+PROGRAM_COUNTER,
          Current_regs[PROGRAM_COUNTER], MODIFY_BYTES );
        Display_pc = Current_regs[PROGRAM_COUNTER];
        First = Instruction = Get_next_inst();
        num = Inst_length( First );
        if( num > 1 )
          {
            Second = Get_next_inst();
            if( num > 2 )
                Third = Get_next_inst();
          }
        repeat
          {
            if( All_changes )
              {
                One_instruction( Display_pc, First,
                  Second, Third, Temp_str );
                Update_inst();
              }
            REQUEST[request] = SERVER_READY;
            send( request, request, Window_id );
            if( REQUEST[request] == $1 )
              {
                All_changes = 0;
                break;
              }
            if( REQUEST[request] == $2 )
              {
                All_changes = 1;
                break;
              }
            All_changes = 1;
            Redraw_prefetch();
            Redraw_data_prefetch();
            Update_all_registers();
```

```
            Send_to_window( Base_field+PROGRAM_COUNTER,
              Display_pc, MODIFY_BYTES );
          }


        high_bit = High_bit();
        select (high_bit)
          {
            case  0 : Spin();
            case  1 : Switch_state();
            case  2 : Return();
            case  3 : Call();
            case  4 : Flying_leap();
            case  5 : Ncd();
            case  6 : Ncd();
            case  7 : Ncd();
            case  8 : Ncd();
            case  9 : Load_store_alt();
            case 10 : Ncd();
            case 11 : Ncd();
            case 12 : If();
            case 13 : Load_effective_address();
            case 14 : Jump();
            case 15 : Load_store();
            case 16 : Process_regs();
          }
         Update_registers();
      }
}
```

I.2 Code for Mmu process

```
import( Message_set )
()
    sendid      : Pid

  {
    Rec_rep();
    Rec_rep();
    sendid = SRC_ID[Msg];
    Rec_rep();
    From_buss_id = SRC_ID[Msg];
    Rec_rep();
    Rec_rep();
    Receive_id = SRC_ID[Msg];
    Rec_rep();
    Window_id = SRC_ID[Msg];
    Rec_rep();
    repeat
      {
        if( receive(Message, sendid) )
            Handle_msg();
        else
            Destroy( My_id );
        reply( Message, sendid );
      }
  }
```

I.3 Code for Memory Process

```
import( Message_set, IO_descriptor, IO_modes, IO_requests )
()
    f_name   : &char = stack(33)
    mask     : unsigned[32]
    i        : unsigned
    low      : unsigned[32]
    high     : unsigned[32]
    in       : Pid
    out      : Pid
    bits     : unsigned
    bytepos  : unsigned[32]
    lowadd   : unsigned[32]
    highadd  : unsigned[32]
    offset   : unsigned

{
    Rec_rep();
    My_number = WIDTH[Msg];
    low = ADDRESS[Msg];
    high = DATA[Msg];
    Rec_rep();
    in = SRC_ID[Msg];
    Rec_rep();
    Rec_rep();
    out = SRC_ID[Msg];
    Window_id = Rec_rep();
    Rec_rep();
    mask = $80000000;
    for( i=0; i<32; ++i)
      {
        if( mask & low )    f_name[i] = '_';
        else                f_name[i] = '.';
            mask >>= 1;
      }
    f_name[32] = 0;
    Set_current_node( "#me" );
    The_file = Open( f_name, MODIFY, 0, 0 );
    while( receive( Msg, in ) )
      {
        reply( Msg, in );
        bytepos = (ADDRESS[Msg]-low) << 1;
        Load_block( bytepos & ~$7FF );
        bits = WIDTH[Msg];
        if( ACCESS[Msg] )
          {
            if( bits == 1 ) offset = bytepos & $7FE;
            else            offset = bytepos & $7FC;
            The_block[offset] = DATA[Msg];
            The_block[++offset] = DATA[Msg] >> 8;
            if( bits == 2 )
              {
```

```
                The_block[++offset] = DATA[Msg] >> 16;
                The_block[++offset] = DATA[Msg] >> 24;
              }
            Dirty = 1;
            Send_to_window( 1, ADDRESS[Msg], APPEND_BYTES );
            Send_to_window( 0, DATA[Msg], APPEND_BYTES );
          }
        else
          {
            offset = bytepos & $7F8;
            lowadd = The_block[offset+3];
            lowadd = (lowadd << 8) | The_block[offset+2];
            lowadd = (lowadd << 8) | The_block[offset+1];
            lowadd = (lowadd << 8) | The_block[offset+0];
            highadd = The_block[offset+7];
            highadd = (highadd << 8) | The_block[offset+6];
            highadd = (highadd << 8) | The_block[offset+5];
            highadd = (highadd << 8) | The_block[offset+4];
            ADDRESS[Msg] = lowadd;
            DATA[Msg] = highadd;
          }
        REQUEST[Msg] = 0;
        DES_ID[Msg] = SRC_ID[Msg];
        SRC_ID[Msg] = Invalid_id;
        send( Msg, Msg, out );
      }
}
```

I.4 Code for On Board Switch Process

```
    import( Message_set,
            IO_requests )

()
    in_id    : Pid

  {
    Rec_rep();
    in_id = Rec_rep();
    Rec_rep();
    Window_id = Rec_rep();
    Rec_rep();
    while( receive( Msg, in_id ) )
      {
        reply( Msg, in_id );
        Send_to_window( 0, DATA[Msg], HERE_IS_TEXT );
      }
  }
```

I.5 Code for Front Panel "Run" function

```
    import( IO_request,
            IO_requests,
            Field_definitions,
            Active_buttons )

()
{
 Change_window_height( Window_height=FORM_HEIGHT );
 Initialize_image();
 The_active_buttons = PF3_ACTIVE | PF4_ACTIVE | PF5_ACTIVE;
 Redisplay_window();
 repeat
 {
   Flush();
   Requestor = receive_any( Request );
   select( REQUEST[Request] )
   {
     case INPUT_ARRIVED   : Input_arrived();
     case MODIFY_BYTES    : Update_field( Request );
     case APPEND_BYTES    : Scroll_memory_writes(Request);
     case DISPLAY_MESSAGE : Update_inst( Request );
     case SERVER_READY    : Server_ready();
   }

 }
}
```

I.6 Code for Bus process

```
import( Message_set, Pid, Structure )
()
    from_my_board    : Pid
    to_my_board      : Pid
    next_buss        : Pid
    id               : Pid
    low              : unsigned[32]
    high             : unsigned[32]

{
    Rec_rep();
    low = ADDRESS[Msg];
    high = DATA[Msg];
    Rec_rep();
    from_my_board = SRC_ID[Msg];
    Rec_rep();
    Rec_rep();
    to_my_board = SRC_ID[Msg];
    Rec_rep();
    next_buss = SRC_ID[Msg];
    Rec_rep();
    repeat
      {
        id = receive_any( Msg );
        if( id == from_my_board )
          {
            if( REQUEST[Msg] == 0 )
              {
                send( Msg, Msg, next_buss );
                reply( Msg, from_my_board );
              }
            else
              {
                SRC_ID[Msg] = My_id;
                DES_ID[Msg] = Invalid_id;
                send( Msg, Msg, next_buss );
                reply( Msg, from_my_board );
              }
          }
        else
          {
            reply( Msg, id );
            if(DES_ID[Msg] == My_id||SRC_ID[Msg] == My_id)
              {
               if( SRC_ID[Msg] == My_id ) WIDTH[Msg] = 3;
               send( Msg, Msg, to_my_board );
              }
            else if(DES_ID[Msg]==Invalid_id && ADDRESS[Msg]

                    >= low && ADDRESS[Msg] <= high )
              {
```

```
                send( Msg, Msg, to_my_board );
              }
            else
              {
                send( Msg, Msg, next_buss );
              }
          }
      }
}
```

I.7 Code for Serial Line Process

```
import( Message_set, Pid )
()

    mmu_id   : Pid
    gof_id   : Pid
    out1     : Pid
    out2     : Pid
    lower    : unsigned[32]
    upper    : unsigned[32]

  {
    Rec_rep();
    lower = ADDRESS[Msg];
    upper = DATA[Msg];
    mmu_id = Rec_rep();
    gof_id = Rec_rep();
    Rec_rep();
    out1 = Rec_rep();
    out2 = Rec_rep();
    Rec_rep();
    while( receive( Msg, mmu_id ) )
      {
        reply( Msg, mmu_id );
        if(ADDRESS[Msg] >= lower && ADDRESS[Msg] <= upper)
            send( Msg, Msg, out2 );
        else
          {
            send( Msg, Msg, out1 );
            receive( Msg, gof_id );
            reply( Msg, gof_id );
          }
        send( Msg, Msg, mmu_id );
      }
  }
```

I.8 Code for Gofer Process

```
import( Message_set, Pid )
()
        in  : Pid
        out : Pid

{
    Rec_rep();
    My_number = WIDTH[Msg];
    Rec_rep();
    in = SRC_ID[Msg];
    Rec_rep();
    Rec_rep();
    out = SRC_ID[Msg];
    Rec_rep();
    while( receive( Msg, in ) )
      {
        reply( Msg, in );
        send( Msg, Msg, out );
      }
}
```

APPENDIX II

II.1 Assembler mnemonics

```
code   - change output to code file
data   - change output to data file
add    - add the value of two registers
lea    - load effective address
shiftl - shift left
shiftr - shift right
rotl   - rotate left
rotr   - rotate right
ldul6  - load unsigned 16
ldu32  - load unsigned 32
ldil6  - load signed 16
ldi32  - load signed 32
ldalt  - load alternate register
stalt  - atore alternate register
stul6  - store unsigned 16
stu32  - store unsigned 32
stil6  - store signed 16
sti32  - store signed 32
if     - if condition goto
call   - subroutine call
lcall  - long subroutine call
calls  - service cal
lcalls - long service call
jump   - long jump absolute value
hop    - short jump relative value
leap   - flying leap
switch - switch modes
return - return from a subroutine call
or     - bitwise or
and    - bitwise and
@Label - define a label
word   - define a word of data
byte   - define a byte of data
```

Appendix III

III.1 Text file used by program "Set machine"

```
0 #me/Frontpanel          0          0              ;  1 2 3 4
    18 19 6 7 8 9 24 25 10 11 12 13 14 15 16 17 20 21 22 23
1 #me/Alu                 0          0              ;  2 0
2 #me/Mmu                 0          0          1 13 ; 12 0
3 #me/Mem                 0      $4000            15 ; 14 0
4 #me/Mem         $80000000 $80004000            17 ; 16 0
5 #me/Screen              0          0            11 ; 10
6 #me/Buss                0          0            10 ; 11  7
7 #me/Buss                0          0            12 ; 13  8
8 #me/Buss                0      $4000            14 ; 15  9
9 #me/Buss        $80000000 $80004000            16 ; 17  24
10 #me/Gofer              0          0             5 ;  6
11 #me/Gofer              0          0             6 ;  5
12 #me/Gofer              0          0             2 ;  7
13 #me/Gofer              0          0             7 ;  2
14 #me/Gofer              0          0             3 ;  8
15 #me/Gofer              0          0             8 ;  3
16 #me/Gofer              0          0             4 ;  9
17 #me/Gofer              0          0             9 ;  4
18 #me/Mem        $40000000 $40004000            20 ; 21 0
19 #me/Mem        $C0000000 $C0004000            22 ; 23 0
20 #me/Gofer              0          0            24 ; 18
21 #me/Gofer              0          0            18 ; 24
22 #me/Gofer              0          0            25 ; 19
23 #me/Gofer              0          0            19 ; 25
24 #me/Buss       $40000000 $40004000             9 ; 20 25
25 #me/Buss       $C0000000 $C0004000            24 ; 22 6
```

REFERENCES

[1] Bornat Richard, Understanding and Writing Compilers, Macmillan Publishers Limited 1984

[2] Bonkowski Bert, Didur Phyllis, Malcolm Michael, Stafford Gary, Young Terry, Programming in Waterloo PORT, University of Waterloo.

[3] Gosling J. B., Design of Arithmetic Units for Digital Computers, Macmillan Publishers Limited

[4] Knuth Donald E., The Art of Computer Programming, 2nd Ed. Addison-Wesley publishing company Sydney 1973.

[5] Lee Graham, From Hardware to Software: An Introduction to Computers, Macmillan Publishers Limited

[6] McWeeny Paul, Young Terry, The Waterloo PORT programming Input/Output and Window programming, University of Waterloo.

[7] Walker B. S., Understanding Microprocessors, Macmillan Publishers Limited.

Allbook Bindery
91 Ryedale Road
West Ryde 2114
Phone: 807 6026