ORION
Microarchitecture Reference Manual

Third Edition

HIGH LEVEL HARDWARE

ORION

Published by High Level Hardware Limited
Windmill Road, Oxford OX3 7BN

First published 1983
Second edition June 1984
Third edition March 1986

The policy of High Level Hardware is one of continuous development and improvement of its services, and the right is therefore reserved to revise this document or to make changes in the computer hardware and software it describes without notice. High Level Hardware make every endeavour to ensure the accuracy of this document but do not accept liability for any error or omission.

CONTENTS

CHAPTER 1

Introduction to Microprogramming

Microprogramming a computer is not easy. Not only is the format unfamiliar, in comparison with conventional assembly languages, and a detailed understanding of the machine hardware needed, but the microprogrammer is also required to be aware of the consequences of performing several operations in parallel.

The purpose of this document is to describe the working of the various components of the ORION hardware architecture, that is, the basic hardware as opposed to the architecture that is presented to the user by the combination of hardware and microprogram, such that the reader will be able to begin writing new microcode. Along the way, although predominantly concerned with illustrating the interaction of the hardware and the 'standard microcode' which implements a stack machine, we will draw attention from time to time to features that can be exploited in a number of different ways and which may be useful for new applications.

1.1 Overview of the ORION architecture

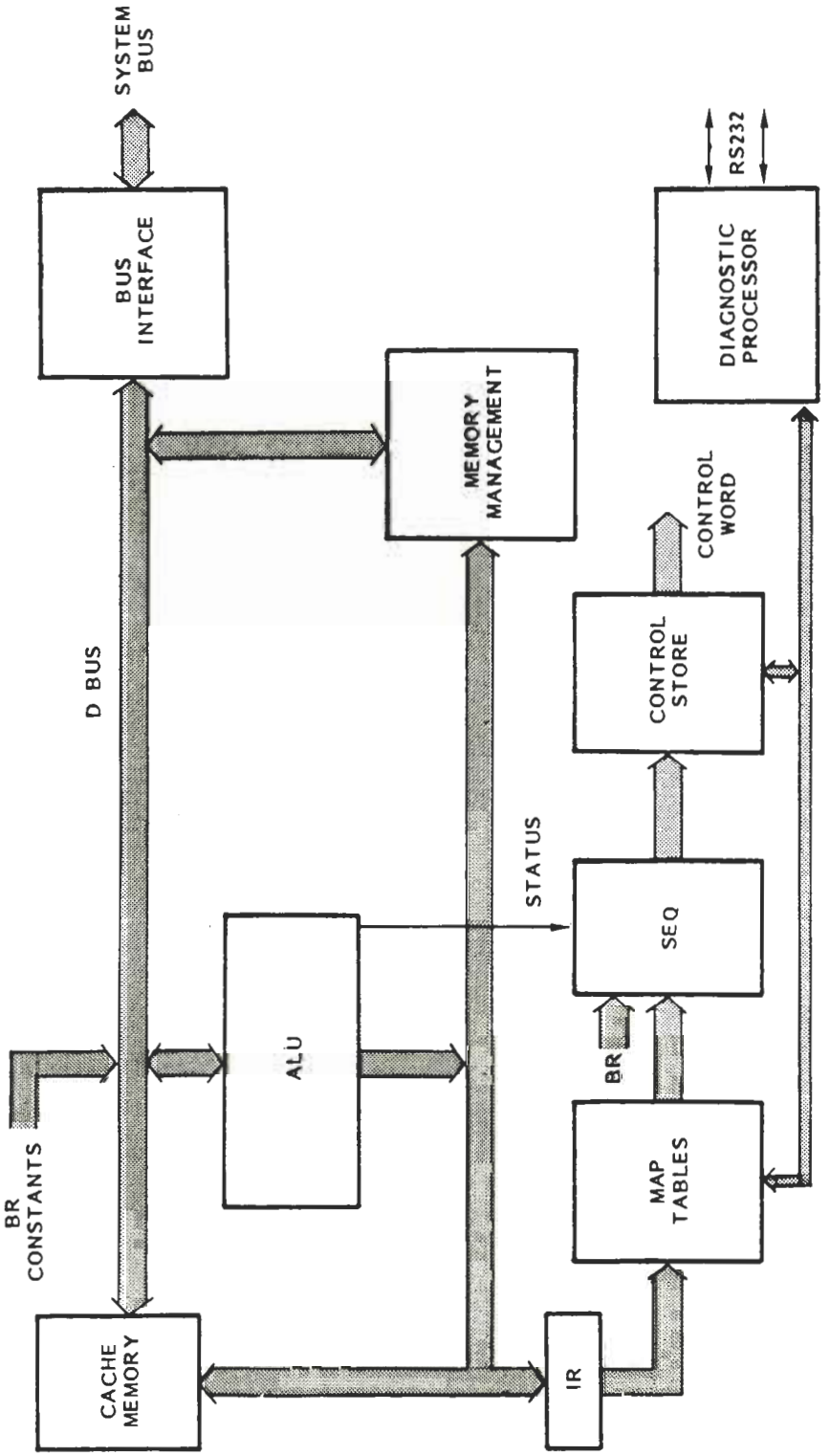
The ORION hardware consists of the following main components

Control store
Microprogram sequencer
Map tables
Arithmetic and logic unit (ALU)
Cache memory
Virtual memory translation buffer
Main memory
I/O subsystems
Diagnostic microprocessor

Of these, everything except the main memory and I/O subsystems is considered to be part of the ORION Central Processing Unit (CPU), depicted in Figure 1.1. The CPU is built around bit-slice microprocessors with a microprogrammed control unit; all the major data paths and functional units are 32 bits wide.

Microprogramming has the advantage of greatly simplifying the hardware of the CPU by replacing large amounts of irregular logic with a highly regular memory (the control store); it also affords the greatest possible flexibility in the application of the machine. The control store is implemented using high speed read/write memory (RAM) and its contents can therefore be changed to optimise the high level architecture, that is, the architecture seen by the end user. Indeed, it is possible to make the machine present several different high level architectures concurrently with suitable programming of the control store.

Figure 1.1 The ORION CPU.



When the machine is in operation, the CPU components must be provided with control information at every tick of the CPU clock (every 125-200ns). This information is read from the control store on each tick and distributed throughout the CPU. Thus, for example, the ALU receives three bits of information to select a function to perform, whilst three more bits determine what should be done with the result.

1.2 Form of the control word

There are two extreme forms of microprogrammed control, known as horizontal and vertical. In the horizontal form, a wide control word (of 40-100 bits) is used and for every unit in the CPU a group of bits (known as a field) provides control information exclusively for that unit. This gives maximum flexibility at the expense of a relatively costly control store, since there are no restrictions imposed by the control word on what operations may be performed in parallel by the different units. A disadvantage is that the microprogrammer may be presented with a bewildering selection of possible operations.

In contrast, in the vertical scheme, the hardware designer decides in advance which combinations of functions are deemed to be useful and encodes them into as small a number of bits as possible. This allows the control store to be of modest width but restricts severely the options available to the microprogrammer. Additionally, the encoded control word must be decoded on every clock tick; this takes time and so can result in reduced performance. In compensation, the greatly reduced number of possible operations simplifies the task of microprogramming.

As with most microprogrammed machines, ORION uses a control word which is partly horizontal and partly vertical. As far as possible the word is horizontal, but many of the infrequently used functions and several mutually exclusive ones are encoded into vertical fields within the horizontal word. For example, when selecting between several sources for a particular data path, the machine operation would be undefined (and could lead to hardware failure) if more than one source were selected simultaneously. Instead of having a separate bit in the word for each source, the source to be selected is encoded as a binary number in a single field. No generality is lost and only a very small delay is incurred in performing the decoding.

The ORION control word is 64 bits wide, divided into 18 fields plus a parity bit.

1.3 Microprogram sequencer

In order that the CPU may perform the more complex high level functions (such as floating point multiplication), the microinstructions held in the control store must be accessed and executed in an appropriate sequence. The CPU component known as the sequencer is dedicated to the task of selecting microinstructions; on every tick of the clock it generates the address of a new instruction to be fetched from the control store.

The microprogram sequencer is quite a complex unit and is itself controlled by several fields in the microinstruction. It allows groups of microinstructions to be executed sequentially or conditionally, repeated in loops, or called as subroutines.

1.4 Map tables

In normal operation the microcode does not consist of a single large program. Rather, it consists of a set of very short programs, each dedicated to the execution of a single machine level instruction, together with a set of utility subroutines shared by several instructions. As each machine level instruction is processed, microprogram control must be passed to the appropriate control store address. The function of the map tables is to provide a mapping between machine level instructions (opcodes) and the relevant control store addresses. An opcode to be executed is loaded into the instruction register which addresses a map table. Then the data found in the selected map table entry is passed to the microprogram sequencer which treats it as a control store address and transfers control to that address.

Several map tables are provided which makes it possible to have several instruction sets in the machine at once. Each instruction set is decoded by a different map table, selected by another part of the instruction register, and the instruction set to be used can be chosen by operating system software when a process is dispatched.

1.5 Arithmetic and logic unit

The arithmetic and logic unit (ALU) is the section of the CPU where most of the actual data manipulation takes place. The ORION ALU is based on the Advanced Micro Devices Am2901C bipolar bit slice microprocessor. This device processes data only four bits at a time. However, by connecting several together, an ALU of any desired width may be constructed. In the case of ORION eight devices provide a 32-bit word length and can be programmed to perform one of three arithmetic or five logical functions; these operations take one machine cycle and are the basic building blocks from which more complex operations are composed. The bit slice processors also contain a one bit shifter, a special register used in compound operations such as multiplication and division and a number of high speed data storage registers. The latter, since they are intimately connected with the data processing circuits, are considered to be a part of the ALU, in contrast to the other storage elements in the CPU.

In addition to the Am2901Cs, the ORION ALU contains a byte manipulation unit to facilitate operations on data of size less than the 32-bit word length. Since the ORION memory system is organised as 32-bit words, the byte unit is used to align and mask the incoming word when, say, only a single byte is required. As will be seen later, the byte manipulation unit can also perform an important role in the decoding of high level abstract-machine instructions.

A facility is provided to allow frequently used constant numbers to be embedded directly in a control word and used as ALU data. Although only twelve bits are reserved in the control word for this

function, sign extension is provided, and the byte manipulation unit can also be applied to extend the range available in a single cycle.

1.6 Cache

Most high performance computers provide a cache memory in the CPU in order to reduce the number of references which must be made to the much larger, but slower, main memory. Unlike a conventional machine in which the cache is designed on the assumption that memory references take place to fairly random locations, the ORION cache is designed to be most effective when the machine is running a modern procedural language. With such languages there is considerable 'locality of reference' and a relatively simple design of cache suffices to provide a high 'hit rate'.

The ORION cache consists of a number of banks of fast memory addressed by registers which allow both random access and the push and pop operations required of a stack. The standard microcode uses a section of the ORION cache to hold the top of the stack on which both variable allocation and expression evaluation take place. In this way the most recently used items, which tend to be close to the top of the stack, are to be found in the cache. Since the cache has several independent segments, it can be used for a variety of other purposes such as to provide a large, fast bank of 'conventional' registers.

The design of the cache has been optimised to allow data to be moved in and out of memory at the maximum rate the memory system will permit; it can provide one word to the ALU per clock cycle.

1.7 Virtual memory

The virtual address translation buffer is the component of the CPU which, in conjunction with microcode, allows the physical memory of the ORION system to be distributed amongst several tasks.

Each time a reference to memory is to be made the address provided by the program, in the form of a virtual address, is translated into a physical address before being sent to the memory system. The address translation process allows the hardware to check for addresses which are outside the limits set by the operating system for that task; errant programs can thus be detected before damage is done to other software in the system.

Support is also provided in the hardware for 'demand paging', the process which allows programs to occupy a large amount of virtual memory without actually consuming too much of the physical store. Those parts of a program which are not in physical memory are stored temporarily on disk, to be loaded into memory on demand.

1.8 Principal data paths

There are two principal 32-bit data paths in the CPU known as the D bus and the AY bus. The D bus interfaces directly to the system bus and to the cache memory and provides the input data to the ALU. Constants direct from the control store appear on the D bus, as do physical addresses from the translation buffer. The AY bus carries

results from the ALU to the special registers which address the cache memory, the map tables, and the virtual memory translation buffer. These two busses are completely independent in order to allow as many units as possible within the CPU to function in parallel. However, a path exists to transfer data direct from the AY bus to the D bus so that results from the ALU may be stored in the cache or main memory, or be recirculated through the ALU for further processing.

The CPU is connected to the memory and I/O subsystems via the system bus. This is a 32-bit multi-master synchronous bus which carries both addresses and data at rates of up to 32 Mbytes/sec. At the end of each transaction arbitration logic transfers control to the highest priority device requesting use of the bus. Priority is determined by physical position on the bus.

1.9 Main memory

The ORION main memory is accessed via the system bus. It consists of one or more modules each providing either 512 Kbytes or 2 Mbytes of storage. The number of modules allowed in a system is limited only by the number of system bus slots available. Each memory module is organised as double words each of 64 data bits together with two parity bits. Such an organisation improves the performance of the system by allowing two words to be accessed in a single memory operation. The data is transferred over the system bus on two successive bus cycles. The bandwidth of the memory system is approximately 16 Mbytes/sec.

Memory operations take several clock cycles to perform and each step in the operation is controlled by a field in the microinstruction. This gives great flexibility and allows complex read/modify/write operations to be performed when required.

1.10 I/O subsystems

The I/O subsystems in ORION offload from the CPU a great many of the low level functions which must be performed in order to control peripheral devices. This allows a higher level and more uniform interface to be provided to the CPU and removes the need for a sophisticated interrupt structure. Each subsystem includes a conventional microprocessor which can issue low level commands to devices, respond to time critical interrupts, and perform error recovery and diagnostic functions. Care has been taken to ensure that the microprocessor does not become a bottleneck when dealing with fast peripherals such as high performance disks. In such cases, a direct memory access (DMA) channel is provided between the peripheral and the ORION memory system, allowing data transfers to take place at the full rate of the device.

I/O subsystems connect to the system bus and are accessed by the CPU as memory mapped registers. All I/O subsystems contain an identification register which allows the system, at the time of initialisation, to determine the nature and number of all peripherals and memory modules. This allows software to reconfigure itself if devices are added or removed temporarily because of a hardware failure.

1.11 Bootstrapping and the diagnostic microcomputer

Since the control store and map tables in the CPU are implemented entirely in RAM the CPU is incapable of any operation when power is first applied. A microcomputer is embedded within the CPU in order to surmount this problem. This microcomputer, which is known as the diagnostic processor (DP), has access both to the control store and the map tables and can load an initial microprogram to start the system. In addition, it has access to many parts of the CPU allowing it to perform diagnostic functions both at power on and when there is a suspected hardware failure. In the latter case diagnostic programs are down-loaded from one of the I/O subsystems. A serial link connects the DP to an I/O subsystem microcomputer or, in the case of a severe system failure, to any other computer equipped with a standard RS-232C serial interface. A second serial interface is available to allow a terminal to be attached directly to the DP for diagnostic use.

During development of new microcode the DP can be used to step though portions of the new code. It can trace the flow through the code and provide a symbolic disassembly to assist with debugging.

During normal operation of the system the only role of the DP is to load or unload portions of the control store or map tables dynamically on command from the main CPU. However, should a parity error be detected in the control store or map tables the DP halts the machine and flashes a front panel indicator. Such a condition is regarded as fatal and full fault diagnosis is necessary before normal operation can resume.

1.12 References

As mentioned earlier, the CPU incorporates several parts from the Advanced Micro Devices Am2900 family. Those unfamiliar with these parts are referred to

> Bipolar Microprocessor Logic and Interface Data Book, 1983, published by Advanced Micro Devices, Sunnyvale, California.

which contains the definitive description of the Am2900 family in Chapter 5; the relevant parts are the Am2901C and the Am2910.

CHAPTER 2

The Microcode Development Tools

Before we can present examples of real microprogramming we must first take a look at the development tools which will be used, and define informally the syntax and semantics of the microassembly language. That is the purpose of this chapter; more formal definitions will be found in the appendices.

Many of the ideas involved in microprogramming will be familiar from conventional assembly languages and we assume here an understanding of concepts such as relocation, linkage, and two pass assembly. Thus, microprograms are assembled with a microassembler (ms) to produce relocatable modules. These are linked using a microlinker (**ml**) to form an absolute load module ready for loading into the control store and map tables. The linker resolves inter-module references and fills in map table entries to point to the appropriate addresses in the control store. It has the ability to search subroutine libraries and load any required modules. The production and maintenance of libraries is facilitated by the microlibrarian (mb).

Ms is a general purpose microassembler which reads a definition of the microinstruction format from a file at the start of an assembly. The standard definitions as used throughout this manual are given in Appendix A.

2.1 Microcode syntax

A microprogram consists of a series of lines of text called statements. Within each statement the characters are grouped to form symbols, operators, and numbers. Symbols fall into two main types; those built into ms or defined by the configuration file, and those created by the programmer. The built in symbols include the names of the fields in the control word, and the mnemonics corresponding to the possible values which may be taken on by those fields. The user defined symbols are used as program labels to identify points in the code which are the targets of control transfer instructions and as mnemonic names for numeric values. As there is a very large number of built in symbols, many of which are used only rarely, a convention has been adopted in order to distinguish them from user defined symbols. Built in symbols are written entirely in upper case, and user defined symbols are normally in lower case. Symbols and numbers are delimited either by operators, or by space and TAB characters which may be used freely to enhance the layout.

Statements may be divided into a number of categories of which only three are of importance here. These are symbol definitions, microinstructions, and map table entries. Any statement may be labelled to identify the addresses of particular points in the code. The text may also include comments which are ignored by the assembler.

2.2 Symbol definitions

Definitions provide symbolic names for numeric and other values which will be used later when the names are encountered in other statements. A typical example of a symbol definition might be

```
foo     =       37
```

which gives the value 37 to the symbol foo. The purpose of definitions is to increase readability and localize the occurrence of 'magic numbers'. Symbols must be defined before they are used. However, ms is a two pass assembler and it is usually adequate to ensure that all symbols are defined by the end of the first pass.

A label is a special case of a symbol definition in which the symbol becomes defined by appearing at the beginning of a statement and followed by a colon. The value assigned to the label is the address of the next microinstruction. Labels that begin with a letter remain in scope for the whole of the assembly and will be included in the symbol table. Labels that begin with a commercial at sign (@) are local, are in scope only between the surrounding pair of ordinary labels, and are not included in the symbol table.

2.3 Microinstructions

Microinstructions cause code to be generated which will subsequently be loaded into words in the control store. An example of a typical microinstruction is

```
CONT   ZB ADD CIN RAMF B=R3
```

Although the precise meaning is unimportant at this stage, this instruction will, when executed, cause the contents of register R3 in the ALU to be incremented. Every microinstruction must define the values of all the fields in the control word, but in general a majority will not be of interest in any particular instruction and ms provides 'harmless' default values for fields which are not given values explicitly. The default values are defined in the configuration file and are chosen to ensure that the state of the machine is not changed in any way by a default field. This frees the programmer to think only about those parts of the machine which are relevant to the operations to be performed by a particular instruction.

Each field may be specified in one of two ways. Usually, it suffices to give just the mnemonic for the desired value of the field. Occasionally it is helpful, and sometimes necessary, to use the combination

```
<fieldname> = <fieldvalue>
```

Thus, in the above example, the B register field is specified in this long form whilst all the other fields are specified in the short form. The long form is required for the A and B register fields since the same mnemonic field values (the names of the ALU registers) are used for both and there is no way for the assembler to distinguish the two. All other fields have unambiguous mnemonic values so that the field name can be inferred from the value.

2.4 Map table entries

Map table entries are written in the form

```
ENTRY <expression>
```

and cause the initialization of one entry in the map tables. The value of <expression> determines which entry is to be initialized, the value to be stored in the entry after the code has been linked being the address of the following microinstruction. In practice, this means that to define a machine level instruction a sequence such as the following (in which the details of the instructions are irrelevant at this stage) is used

```
ENTRY plus

        CONT   DZ SHR1  OR   RAMA A=ir0 B=ir0  DECCA  LDIR
        CONT   DZ CSH   OR   RAMF B=acc
        CJV    DA CSH A=acc  ADD  RAMF B=acc
        CONT   ZB  SUBR  RAMA A=acc B=sp  CWR
```

In this example, plus is a symbol whose value is the opcode for a machine instruction called plus which adds the two top items on a stack and replaces them by the result. After assembly and linking the map table entry corresponding to the opcode plus will point to these four instructions and control will be transferred here whenever a plus instruction is decoded. No other action is needed on the part of the microprogrammer in order to arrange this. The CJV in the penultimate instruction transfers control to the microcode for the next opcode after the plus has been executed.

This example is typical of real microprograms; a short sequence of microinstructions in isolation dedicated to a particular higher level operation.

2.5 Layout

Aside from the requirement that statements be written on a single line there are few restrictions on layout. Fields within an instruction may be given in any convenient order. However, in the interests of program readability it is wise to stick closely to a convention. Combinations of fields often occur together and consistent ordering of the fields makes these combinations instantly recognisable, thus enhancing the readability. The standard convention is to place the sequencer opcode first in an instruction, separated from the rest of the instruction by one TAB stop. The remaining fields are separated from each other by either one or two spaces. One space is used when adjacent fields are in some way related, otherwise two. Although it is possible to allow the sequencer field to take a default value (CONT), or to place it anywhere in the instruction, it is considered to be of such significance that it is always written explicitly at the start of each instruction. This corresponds to the fact that every microinstruction must explicitly nominate a successor.

Instructions are normally indented by one TAB stop from the left margin in order to distinguish them from labels, entry points and assembler directives. When a section of code forms the body of a loop it is indented by one additional TAB stop from the surrounding

instructions to make this more apparent. The importance of this will become clearer when we consider the control store pipeline.

Comments should be many and voluminous since opportunities for writing obscure microinstructions abound. It is considered acceptable to employ programming 'tricks' at the microcode level since the system performance will be proportional to the number of microinstructions executed. Shaving a single instruction off a critical piece of code may make several percent of difference to the overall speed. Block comments of several lines between related groups of instructions are better than terse comments on each instruction. This is partly because the lines are too long if a comment is added, but mostly because such comments usually say what an instruction does rather than why. The former can be deduced from the instruction itself, the latter cannot.

Several ways of introducing comments are provided of which the preferred method is to enclose them between /* and */ pairs. This makes it possible to create files of symbol definitions which can be included in both microcode and high level language (for example BCPL) programs, thus avoiding a common mistake whereby the microcode uses a different value from the rest of the system for some 'constant'. This method is also consistent with the C preprocessor which can be used to add macro facilities.

Appendix E lists some of the microcode for the ORION standard instruction set that supports the C language and the OTS operating system. This is provided both for reference and as an example of layout and programming style. The operation of the standard instruction set is outlined in Chapter 10.

CHAPTER 3

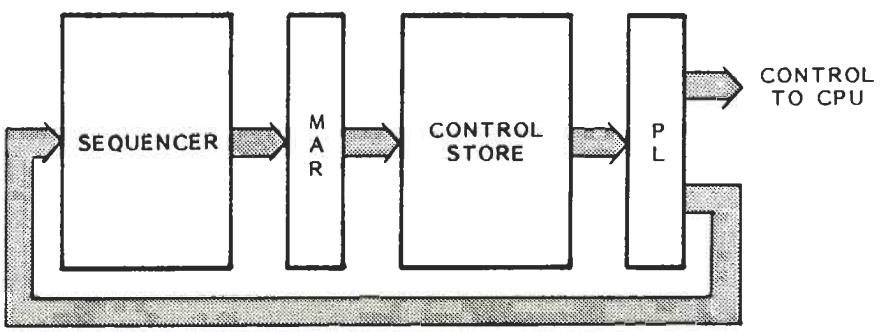
The Microprogram Sequencer and the Control Store

The microprogram sequencer controls the order in which microinstructions are fetched from the control store and subsequently executed. The major component in the sequencer is the Am2910, a bipolar LSI device designed specifically for this application. The control store consists of up to 32K 64 bit words of static RAM divided into 4K word pages and capable of cycling once per 125ns.

3.1 The control store

The control store must provide one microinstruction to the CPU at the start of each machine cycle. The instructions read out from the memory array are latched in a 64 bit register known as the pipeline register (PL) in order to allow the memory devices themselves to start accessing the next instruction. The instruction in PL is known as the 'current instruction'. The sequencer is itself controlled by fields within the microinstruction. It starts its job of generating a new address as soon as the current instruction has been latched in PL. In general, the address which is generated by the sequencer will be conditional upon the result of some status signal in the CPU and so additional time is often needed before a new address is available. If the address were taken direct from the sequencer to the memory devices in the control store then the delay of the sequencer in generating a new address would be in series with the delay of the memory devices in accessing the data. Although this is the organization used in many microprogrammed CPUs, it requires either very high speed (and hence expensive) memory devices to be used, or else a reduction in the speed of the CPU. Instead, in ORION, an additional register, the microinstruction address register (MAR), is placed between the sequencer and the memory devices. This register is loaded at the start of each cycle at the same time as the pipeline register.

Figure 3.1 The sequencer and control store.



This arrangement, which is depicted in Figure 3.1, puts the two delays in parallel since the sequencer can now be computing a new address at the same time as the memories are accessing an instruction. Unfortunately, there is a complication. The sequencer can no longer generate the address of the next microinstruction to be executed since that is already in the microinstruction address register and the instruction is being accessed from the memories. Instead, it generates the address of the next but one instruction. The simplest sequencer function is CONT, for continue, which causes sequential addresses to be generated. CONT is used whenever no explicit transfer of control is needed. So, to illustrate the effect of the pipeline, consider the following sequence of instructions which form the simplest possible microprogram; it does nothing, sequentially!

```
@1:     CONT
@2:     CONT
@3:     CONT
@4:     CONT
@5:     CONT
@6:     CONT
```

The instructions have been labelled for reference only. Ignore the problem of initialization and assume that instruction @1 is in the pipeline register (and being executed) and that the address of @2 is in the microinstruction address register. This implies that on the previous cycle @2 was the address generated by the sequencer. The sequencer is now seeing the CONT at @1 and so generates an address one bigger than it did on the previous cycle, namely @3. At the end of the cycle both registers are reloaded; the pipeline register then contains the instruction at @2, and the address register the address @3. Since the new instruction is also CONT this process will be repeated with the sequencer generating addresses @4, @5, @6 and so on.†

3.2 The sequencer

Most of the sequencer functions involve some form of control transfer and we will begin with the simplest of these, CJP, conditional jump. Consider the following sequence of instructions

```
@1:     CONT
@2:     CJP     @6
@3:     CONT
@4:     CONT
@5:     CONT
          .
          .
          .
@6:     CONT
@7:     CONT
@8:     CONT
```

The instruction at @2 tells the sequencer to transfer control to the

---

† The CIN pin of the Am2910 is wired permanently high to ensure that the pc is always incremented.

instruction labelled @6. The @6 in this instruction represents the address of the instruction labelled @6. After assembly and linking the relocated address of this label will be stored in a twelve bit field in the instruction known as the branch field (BR). During the CJP instruction the sequencer takes the number in the branch field and sends it as the next address to the microinstruction address register. It also updates its internal microprogram counter accordingly.

When the CJP at @2 is being executed the CONT at @3 is already being fetched and will be loaded into the pipeline register at the start of the next cycle. Thus, even though control is being transferred by the instruction at @2, the instruction at @3 will still be executed next. This means that the programmer must specify control transfers one instruction before the point at which the actual transfer is required.

As its name suggests, CJP is conditional and we must briefly consider conditional instructions in general here, since, as will be seen, most of the sequencer functions are conditional. A full discussion is deferred until a later section. A field in the instruction known as CC (condition code) selects which of a number of status conditions is to be tested. An example is the zero flag (Z) from the ALU which, when true, indicates that the result of the current ALU operation is zero. For each condition code there is a complementary one which is its inverse. Thus the condition code NZ, or non-zero, is the complement of Z and is true unless the ALU result is zero.

We can modify the previous example so that the branch becomes conditional on a zero result as follows

```
@1:     CONT
@2:     CJP     Z,@6
@3:     CONT
@4:     CONT
@5:     CONT
          .
          .
          .
@6:     CONT
@7:     CONT
@8:     CONT
```

If the result is zero then the sequence will be as before. However, if it is non-zero then the CJP will behave in exactly the same way as CONT and the sequence of execution will be @1, @2, @3, @4, @5. The comma between Z and @6 is treated like white space by the assembler and is used simply to emphasize the close link between the two fields in the instruction.

Going back to the first example we see that no condition code was specified. As with all fields, the assembler provides a default setting when none is given explicitly and in the case of CC the default is T (true), a condition which is guaranteed to be true. This means that unless a condition code is specified then a conditional sequencer func-

tion is forced to occur unconditionally.†

The sequencer contains a five level push down stack which is used to allow microcode subroutines to be called. The stack is used to save the state of the microprogram counter so that it may be restored when the subroutine returns. A pair of functions CJS (conditionally call subroutine) and CRTN (conditionally return from subroutine) is provided for this purpose. CJS is similar to CJP except that the old microprogram counter is pushed on the stack before control is transferred if the condition is passed. This address is subsequently popped off the stack again by a successful CRTN to transfer control back to the caller. Thus in the following sequence

```
@1:     CJS     @5
@2:     CONT
@3:     CONT
@4:     CONT
          .
          .
          .

@5:     CRTN
@6:     CONT
```

the flow will be @1, @2, @5, @6, @3, @4 etc. Again we see the one instruction delay before the control transfer and in particular observe that the address saved on the stack is actually @3, since that is the address which would have been generated in the absence of the CJS.

It is the user's responsibility to ensure that the five word stack never overflows. Pushing an item onto a full stack causes the top item to be overwritten without warning. Likewise, popping an empty stack yields rubbish.††

The sequencer contains a twelve bit register which can be used either as a temporary store for an address or as a counter for loop control. For either purpose, a value must first be loaded into the register. This can be done using the LDCT (load counter and continue) function which loads the twelve bit value from the branch field into the counter, but is otherwise the same as CONT. Thus the instruction

```
LDCT   42
```

will load the value 42 into the counter. The value to be loaded may be any numeric expression involving literal and symbolic constants and variables. Such expressions will be evaluated by the assembler and, assuming the resulting value can be represented in twelve bits, will be

---

† The CCEN input of the Am2910 is wired permanently low so that the condition is always tested internally by the Am2910. Hence the need for the T condition code.

†† The FULL output of the Am2910 is unused.

stored in the branch field of the instruction.†

Microcode loops may be constructed in many different ways. Perhaps the simplest is

```
@1:             CONT
                  .
                  .
                  .
                CJP     <cc>, @1
                CONT
@2:     CONT
```

in which control is passed back to @1 if the condition <cc> is true, and falls through to @2 if it is false. However, the sequencer has a rich set of functions specifically designed to allow great flexibility in the construction of loops.

The PUSH function is used to set up loops in which the address of the first instruction in the body of the loop is saved on the top of the sequencer stack. This has the advantage that the branch field is not needed on the loop terminating instruction in order to transfer control back to the top of the loop. PUSH also conditionally loads the counter with the value from the branch field. This value can then be used to control the number of loop iterations to be performed. The sequence

```
        PUSH    <exp>
        CONT
@1:             CONT
```

causes the counter to be loaded with the value of the expression <exp> and at the same time pushes the address of the instruction at @1 onto the stack. Thus the instruction at @1 will be the first instruction of a loop.

PUSH is conditional; if the condition is false then the counter is not loaded. This is sometimes useful because, as will be seen later, the branch field serves also to provide constants as data to the ALU. If such a constant is required on the instruction containing the PUSH then the counter cannot be loaded (except in the unlikely case that the same value is required for both). If the counter is loaded before the PUSH then the condition code F (guaranteed false, the complement of T) must be specified on the PUSH to prevent it from being reloaded. As an example, consider

```
        LDCT    <exp1>
        PUSH    F       <exp2>
        CONT
@1:             CONT
```

in which the loop count will be set to the value of <exp1>, and the value of <exp2> is available to the ALU. Alternatively, because of the pipeline delay, the counter can be loaded on the instruction following the PUSH as in

---

† The RLD input of the Am2910 is wired permanently high.

```
        PUSH
        LDCT    <exp>
@1:             CONT
```

where, as before, the first instruction in the loop body will be @1.

The loop control function most commonly used with PUSH is RFCT. RFCT tests the counter for zero and if it is pops the stack and otherwise behaves as CONT. If the counter is non-zero, then RFCT decrements the counter and transfers control to the address on top of the stack, without popping it. A simple loop using RFCT would be

```
        PUSH    <exp>
        CONT
@1:             CONT
@2:             RFCT
@3:             CONT
@4:     CONT
```

in which the body of the loop is the three instructions @1, @2, and @3. Since the counter is tested for zero before being decremented, these three instructions will be executed <exp> + 1 times before control falls through to @4.

A similar function is RPCT which behaves like RFCT except that if the counter is non-zero then the address is taken from the branch field instead of from the stack. In this case no PUSH is needed to set up the address. When the loop terminates the stack is not popped since it has not been used for the address. As an example, consider

```
        LDCT    <exp>
@1:             CONT
@2:             RPCT    @1
@3:             CONT
@4:     CONT
```

in which the loop body is again @1, @2, and @3. An LDCT is used to load the counter with the value of <exp> without pushing an address onto the stack. As in the previous example, the loop will be executed <exp> + 1 times.

Often it is necessary to execute a loop a variable number of times, or to continue until an arbitrary condition is satisfied. In such cases the sequencer counter cannot be used since it is loaded from the branch field which can only generate constants. Instead, a register in the ALU is used, and the terminating condition can then be any of the testable condition codes. The sequencer function LOOP is useful in such cases. It tests the condition code, and if true it pops the stack but otherwise behaves as CONT; if false it transfers control to the address on top of the stack without popping it. In the following example we assume that a portion of the instruction @2 which is not shown will generate the condition being tested (as will be described later, and contrary to expectation, it is normally necessary to test a condition on the instruction which generates it).

```
        PUSH
        CONT
@1:             CONT
@2:             LOOP    <cc>
@3:             CONT
```

The loop body consists of @1, @2, and @3. These instructions will be repeated until <cc> is true.

To break out of a loop prematurely when the top of the stack is being used to hold the loop start address, we cannot simply use CJP since that would leave the stack unbalanced. Instead, a function CJPP (conditional jump and pop) is available which is equivalent to CJP except that the stack is popped if the condition is passed.

```
        PUSH    <exp>
        CONT
@1:             CJPP    <cc>, @5
@2:             CONT
@3:             RFCT
@4:             CONT
@5:     CONT
```

Here we see a loop in which the body consists of the four instructions @1 to @4 which will be executed at most <exp> + 1 times. However, if on any iteration the <cc> is true on instruction @1, the control flow will be @1, @2, and then to @5, the loop exit point, with the stack correctly balanced.

CJPP can also be used to abort a subroutine if a serious error condition is detected. In such a case we may want to transfer control to an error handler which will never return control to the original caller. The pop will throw away the saved return address.

The final and most complex of the loop control instructions is TWB (three way branch). This single function tests two different conditions and transfers control to one of three different addresses all in one instruction. If the condition code selected in the instruction is true then the stack is popped, the counter is decremented unless it is zero, and control falls through sequentially as with LOOP. Otherwise, if the counter is zero then the stack is popped and control is passed to the address specified in the branch field. Otherwise the counter is decremented and control passes to the address on the top of the stack. The following example illustrates this.

```
        PUSH    <exp>
        CONT
@1:             CONT
@2:             TWB     <cc>,@5
@3:             CONT
@4:     CONT
          .
          .
          .

@5:     CONT
```

The loop body consists of the three instructions labeled @1, @2, and @3. If on any iteration of these three instructions <cc> is true, then the control flow will be @1, @2, @3, @5. Otherwise, if the count is exhausted we get @1, @2, @3, @4.

Two further control transfer functions are JRP and JSRP. These are similar to CJP and CJS, except that control is passed to one of two different addresses depending on the condition. If true, control goes to the address specified by the branch field; if false, it goes to the address currently stored in the counter. A previous instruction must have loaded such an address into the counter. For example

```
        LDCT   @1
        JRP    <cc>, @2
        CONT
/* Never fall through here */

@1:     CONT   /* If false */
         .
         .
         .

@2:     CONT   /* If true */
         .
         .
         .
```

An interesting use of these instructions arises when it is required to transfer control on a particular instruction but the branch field is already in use providing a constant to the ALU. By loading the target address into the counter on a previous instruction and specifying JRP or JSRP with a condition code of F (guaranteed false), an unconditional transfer can be performed on the required instruction, as shown here.

```
        LDCT   @1
        JRP    F       /* Branch field free */
        CONT
         .
         .
         .

@1:     CONT
```

The only remaining important sequencer function is CJV which is used to dispatch high level machine instructions. It is similar to CJP except that if the condition code is true, the transfer address is taken not from the branch field, but from a look-up table (known as a map table) which maps opcodes onto microinstruction addresses. In addition the three bit microcode segment register is reloaded allowing the transfer to cross a 4K word page boundary. The use of this instruction will be discussed more fully below in the next section, and in Chapter 5.

Two final sequencer functions are JUMP and JZ (jump to address zero). JUMP is entirely equivalent to CJP with a true condition and

may be used to emphasize that a jump is to take place unconditionally. It also finds use in diagnostic microcode which needs to be able to transfer control even if the condition code logic in the CPU is not functioning correctly. JZ is executed only when the CPU is first started and causes initialization of the sequencer stack and microprogram counter. The stack is cleared and the program counter is set to zero. JZ is discussed in Chapter 10.

3.3 Control store segmentation

The ORION architecture allows for up to 32K words of control store. However, the main element in the microprogram sequencer, the Am2910, is only 12-bits wide and can thus only directly address 4K words of control store. In order to exceed this limit a 3-bit microcode segment register is provided which supplies three additional address bits to the control store, making up the full 15-bit address required to select a word from the 32K range. The segment register is cleared at initialization so that the microcode always begins execution in segment zero. After this the only operation which can affect the contents of the segment register is the CJV sequencer function. The CJV instruction is conditional and if the condition code selected is false then no special action is performed and the instruction is equivalent to CONT. However, if the condition is true then control is transferred to an address supplied by the map tables. The map tables provide a full 15-bit microinstruction address, the address of the first microinstruction to be executed in order to perform the machine instruction which has just been decoded. The sequencer takes the low order 12-bits to reload its microprogram counter whilst the high order 3-bits are loaded into the segment register.

There are few practical difficulties arising from the segmentation of the control store. Since most microcode consists of small sequences of instructions each dedicated to the execution of a single high level machine instruction, the only significant restriction presented by this control store segmentation is that the microcode used to implement any single high level instruction must reside entirely within one segment of the control store. If a subroutine is to be used in more than one segment, then it is necessary to include a copy of the subroutine in each segment since it cannot be called across a segment boundary. This can be accomplished automatically if subroutines are stored in a library which is searched by the microlinker ml when microcode is linked. It is usual to link the microcode for each segment independently so that libraries will be searched automatically at that point. Indeed, the microlinker will issue a warning if the size of the linked image exceeds 4K words. This is normally an error condition.

3.4 Condition codes

The condition code field in each microinstruction selects one of sixteen status signals from several parts of the CPU to be sampled by the sequencer. Either the true state or the complemented state of the selected signal can be chosen giving 32 possibilities in all. There is no 'status register' in the CPU so that most conditions must be tested on the cycle which generates them or they will be lost. However, the LC

Table 3.1 Summary of sequencer functions.

Mnemonic	Description	Current counter value	Counter function	Condition code			
				False		True	
				Addr	Stack	Addr	Stack
JZ	Jump zero	X	Hold	0	Clear	0	Clear
CJS	Conditional jump subroutine	X	Hold	PC	Hold	BR	Push
JUMP	Unconditional jump	X	Hold	BR	Hold	BR	Hold
CJP	Conditional jump	X	Hold	PC	Hold	BR	Hold
PUSH	Push and conditional load counter	X	Note 1	PC	Push	PC	Push
JSRP	Conditional jump subroutine	X	Hold	Counter	Push	BR	Push
CJV	Conditional dispatch instruction	X	Hold	PC	Hold	BR	Hold
JRP	Conditional jump	X	Hold	Counter	Hold	BR	Hold
RFCT	Test end of loop	$\neq 0$	Decr	Stack	Hold	Stack	Hold
		$= 0$	Hold	PC	Pop	PC	Pop
RPCT	Test end of loop	$\neq 0$	Decr	BR	Hold	BR	Hold
		$= 0$	Hold	PC	Hold	PC	Hold
CRTN	Conditional return	X	Hold	PC	Hold	Stack	Pop
CJPP	Conditional jump and pop	X	Hold	PC	Hold	BR	Pop
LDCT	Load counter and continue	X	Load	PC	Hold	PC	Hold
LOOP	Test end of loop	X	Hold	Stack	Hold	PC	Pop
CONT	Continue	X	Hold	PC	Hold	PC	Hold
TWB	Three way branch	$\neq 0$	Decr	Stack	Hold	PC	Pop
		$= 0$	Hold	BR	Pop	PC	Pop

Note 1. If condition code is true then load else hold.

bit (see below) allows a single status condition to be preserved for more than one cycle. The following paragraphs describe the condition codes in detail, starting with those which are most frequently used. A summary will be found in Table 3.2.

T (true) is a condition guaranteed to be true. It is provided so that sequencer functions which would otherwise be conditional can be made to execute unconditionally. This is the default case and if no other condition code is specified explicitly in an instruction the sequencer function will be unconditional. Thus the instruction

```
CJS     @1
```

is equivalent to

```
CJS     T, @1
```

and results in the subroutine at address @1 being called unconditionally. The complementary condition F (false) is available but is rarely used.

A group of condition codes relates to the result of the current ALU operation.

Z (zero) is true if the result is zero. The complementary condition NZ (non-zero) is true if the result is non-zero.

C (carry) is true if an addition operation, regarded as unsigned, produces a carry from the most significant bit (MSB) of the ALU. The complementary condition NC (no carry) is true if there is no carry. During subtraction operations the carry output of the Am2901C is inverted so that the C condition is true if there is no borrow. Alternative mnemonics are provided, called BW (borrow) which is identical to NC and NBW (no borrow) which is identical to C to assist the programmer in making his intentions clear.

O (overflow) is true if an addition or subtraction operation, regarded as two's complement, overflows. In this context overflow is defined as the exclusive OR of the carry in and the carry out of the MSB of the ALU. The complementary condition NO (no overflow) is true if there is no overflow.

S (sign) is true if the MSB (the two's complement sign bit) of the result is set. The complementary condition NS (no sign) is true if the MSB is not set. If a two's complement operation overflows, that is the O condition is true, then the sign determined from the S condition is incorrect.

CS (corrected sign) is the exclusive OR of S and O. It gives the true sign of a two's complement result even if overflow occurs. It is useful for comparison operations in which the actual result is unimportant but the correct sign is needed. NCS (not corrected sign) is the complementary condition and is true whenever CS is false.

OD (odd) is true if the value on the AY bus has its least significant bit (LSB) set. Although the AY bus is driven by the ALU output this is not necessarily the same as the result of the current ALU operation. A full discussion of this will be found in Chapter 4.

The complementary condition EV (even) is true if the LSB of the AY bus is zero.

OB (odd byte) is true if the value on the AY bus has bit 26 set. The complementary condition EB (even byte) is true if bit 26 is zero. This condition code and the following one are provided to facilitate byte addressing. Physical addresses are word addresses but bits 26 and 27 are ignored by the hardware. They can therefore be used by microcode to store the two additional bits required to address the four bytes within a word.

OS (odd short) is true if the value on the AY bus has bit 27 set. The complementary condition ES (even short) is true if bit 27 is zero.

In addition to being sent to the sequencer, the selected condition code is also saved in a one-bit memory. This value can be tested on the following cycle by specifying the condition LC (last condition). The complementary condition NLC (not last condition) is available to test the complement of the condition selected on the previous instruction. LC and NLC allow the testing of a condition to be deferred for a cycle which is often useful if the branch field of an instruction is involved in the generation of a condition. Under these circumstances it cannot simultaneously provide the target address for a jump instruction. By deferring the test, the branch field can be used on the next instruction as this example shows

```
CONT   <cc>    /* Branch field in use here */
CJP    LC,@1   /* Test saved condition here */
CONT
```

If necessary, several consecutive LCs can be used to preserve the condition indefinitely.

Because of the instruction pipeline, rather subtle control flow paths are possible when consecutive instructions both carry conditional sequencer functions. Usually a true condition should occur on only one of the two with the other behaving as a CONT. Consider the following example

```
CJP    <cc>,@1
CJV    NLC
CONT
```

Here, the first instruction conditionally branches to @1. If the condition <cc> is false then the second instruction will perform the CJV function since NLC will be true. However, if the condition is true, then the NLC will be false and second instruction will behave as a CONT. One, and only one, of the two conditional functions will occur. If two conditions must be generated on consecutive cycles the LC condition can be used to defer the second test until after the pipeline delay associated with the first.

The remaining condition codes all test very specialized conditions and will be described in the chapters covering the sections of the machine which generate the conditions. However, for completeness, they are summarized in Table 3.2.

Table 3.2 Summary of condition codes

Mnemonic	Name	Description
T	True	Guaranteed true.
F	False	Guaranteed false.
Z	Zero	True if current ALU result is zero.
NZ	Non-zero	True if current ALU result is non-zero.
C	Carry	True if an unsigned arithmetic operation produces a carry.
NC	No carry	True if an unsigned arithmetic operation produces no carry.
O	Overflow	True if a two's complement arithmetic operation overflows.
NO	No overflow	True if a two's complement arithmetic operation does not overflow.
S	Sign	True if the MSB of the current ALU result is set.
NS	No sign	True if the MSB of the current ALU result is not set.
CS	Corrected sign	The exclusive OR of S and O.
NCS	No corrected sign	The exclusive NOR of S and O.
OD	Odd	True if the LSB of the AY bus is set.
EV	Even	True if the LSB of the AY bus is not set.
OB	Odd Byte	True if bit 26 of the AY bus is set.
EB	Even byte	True if bit 26 of the AY bus is not set.
OS	Odd Short	True if bit 27 of the AY bus is set.
ES	Even Short	True if bit 27 of the AY bus is not set.
LC	Last condition	True if the condition code was true on the last cycle.
NLC	Not last condition	True if the condition code was false on the last cycle.
RFLT	Read fault	True if an addressing fault occurred during a memory read.
NRFLT	No read fault	True if no addressing fault occurred during a memory read.
WFLT	Write fault	True if an addressing fault occurred during a memory write.
NWFLT	No write fault	True if no addressing fault occurred during a memory write.
INT	Interrupt	True if the interrupt line is asserted.
NINT	No interrupt	True if the interrupt line is not asserted.
TBP	Tr buf parity	True if a translation buffer parity error occurred.
NTBP	No tr buf parity	True if no translation parity error occurred.
MP	Memory parity	True if a memory parity error occurred.
NMP	No memory parity	True if no memory parity error occurred.
PE	Parity error	True if a translation buffer, cache, or main memory parity error occurred.
NPE	No parity error	True if no parity error occurred.

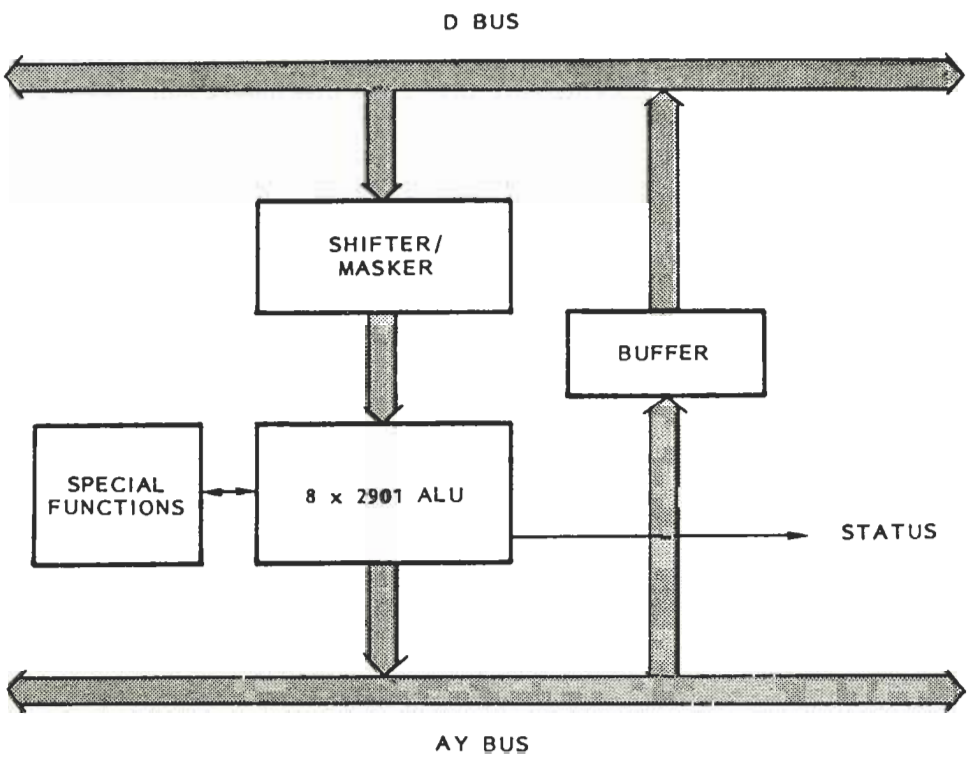
CHAPTER 4

The Arithmetic and Logic Unit

The ALU forms the heart of the CPU. It is where all the actual manipulation of data takes place and where the fastest storage in the CPU resides. A block diagram of the ALU is presented in Figure 4.1.

The ALU is connected between the two main data paths in the CPU. Data enters from the D bus, which connects to the cache memory and the system bus, and results appear on the AY bus. A buffer connects the AY bus back to the D bus to allow results to be stored in the cache or main memory, or to be recirculated through the ALU for further processing. Between the D bus and the Am2901C array lies the shifter. Its functions are to rotate the data passing through by 0, 1, 2, or 3 bytes and then to apply a mask which sets any combination of the shifted bytes to zero. Since its default action is to pass the data through unmodified we can defer a detailed consideration of its action to a later section.

Figure 4.1 The ORION ALU.



4.1 The Am2901C

The Am2901C is a 4-bit bit-slice microprocessor. Eight of these are connected in parallel to form the basis of the 32-bit ALU. Since all eight devices operate in parallel, each performing operations on four bits of the operands and providing four bits of storage for each of the 32-bit registers in the ALU, it suffices to consider only a single Am2901C in order to understand this section of the machine. A block diagram of the Am2901C is shown in Figure 4.2.

Referring to this figure, we see towards the bottom the eight function ALU which operates on a pair of operands labelled R and S to produce the result F. This result can be made available, via the output data selector, on the AY bus. Indeed, this bus is so named because it is driven by the ALU Y output. The carry in line allows a carry to be inserted into the least significant bit of the ALU during arithmetic operations. This line is normally controlled directly by a one bit field in the control word. Although there are eight 4-bit slices, only the carry into the least significant slice can be controlled directly by the microcode.

A number of status signals are shown to the right hand side of the ALU. The line 'F = 0000' is true to indicate a zero result. The corresponding signals from all eight slices are connected together to generate the Z condition code which is true only when all eight slices signal a zero result together. OVERFLOW indicates a two's complement arithmetic overflow. This signal, from the most significant slice, provides the O condition code. F3(SIGN) is the same as the most significant bit of the ALU result. It is available as a separate line so that it can be tested by the sequencer even when the output data selector is not making the ALU result available on the AY bus. This signal, from the most significant slice, provides the S condition code. CN+4 is the carry out of the most significant bit in arithmetic operations. This signal, from the most significant slice, provides the C condition code. The G and P signals provide carry propagation between the slices in arithmetic operations and are of no consequence to the microprogrammer.

The ALU can perform three binary arithmetic operations and five logic operations on the R and S operands. The operation to be performed is controlled by the ALUFUN field in the control word. (We will see later that this operation may be modified by certain 'special functions'.) Table 4.1 details the possible operations.

Addition, subtraction, and reversed subtraction can be used for both two's complement and unsigned operations. The only difference lies in the interpretation of the condition codes. If the operands are unsigned then the O, S, and CS condition codes are meaningless but C is relevant. If the operands are regarded as two's complement then C is meaningless but O, S, and CS are relevant.

Referring again to Figure 4.2 we see that the R and S operands to the ALU are provided by the ALU data source selector. This selector can choose each operand from one of five sources, though as we will see, only eight of the possible combinations are provided, controlled by the ALUSOURCE field of the control word. The direct data input (D)

Figure 4.2 Am2901C block diagram.

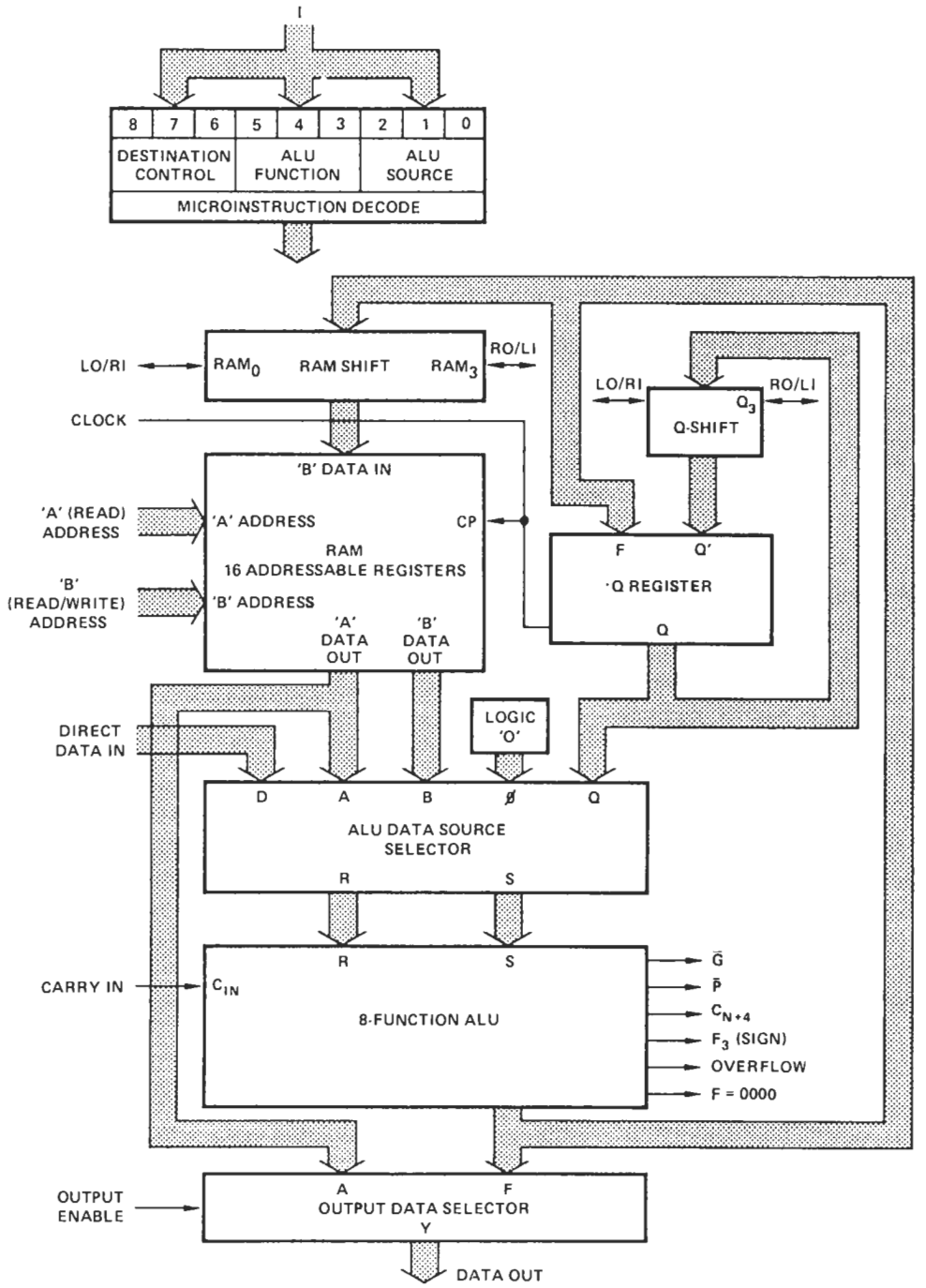


Table 4.1 ALU function mnemonics.

Mnemonic	Function	Description
ADD	R + S	Addition.
SUBS	R - S	Subtraction.
SUBR	S - R	Reversed subtraction.
OR	S OR R	Logical inclusive OR.
AND	S AND R	Logical AND.
NOTRS	(NOT R) AND S	Logical AND of NOT R and S.
EXOR	R XOR S	Logical exclusive OR.
EXNOR	R XNOR S	Logical exclusive NOR.

accepts data from the shifter, which in turn takes its input from the D bus. The zero input allows one of the operands to be set to zero without the need to generate zero external to the Am2901C. The O input takes data from the O register, a special purpose register used mainly during multiplication and division. A and B carry data from the internal register file.

The Am2901C contains a dual port file of sixteen high speed registers. These registers are normally used both as scratch registers in complex microcode sequences, and as the internal registers of the machine being emulated by the microcode. In the standard microcode this includes the stack pointer, frame pointer, and program counter of the abstract 'C' machine. They are not to be thought of as comparable to the general purpose registers of a typical register machine. If such a machine were being emulated on ORION it is more likely that the storage for the general purpose registers would be provided by a region of the cache memory. The register file is addressed by fields in the control word and so register selection is fixed in the microcode. In contrast locations in the cache memory can be selected by addresses determined dynamically at run time.

The term 'dual port' refers to the fact that two of the registers in the file can be accessed simultaneously. Two fields in the control word, called the A and B register fields, each provide four bits to select one of the sixteen registers. The registers can be referred to by the names R0 to R15. However, it is usually desirable to define names of greater mnemonic significance depending on the application. The A and B addresses are entirely independent and can, and often do, select the same register. The data from the selected A and B registers is available at the A and B inputs of the ALU data source selector, which can make it available when required at the R and S

operand inputs of the ALU. Results from the ALU can be returned to the registers. However, when data is to be written to a register, it always goes to the one selected by the B address. The register file is organized such that read/modify/write write operations are possible in a single CPU cycle.

Eight ALUSOURCE functions are available. The mnemonics for these each consist of two letters, the first indicating the source applied to the R input and the second indicating the source applied to the S input. Thus, AB means that the R operand will be the contents of the A register whilst the S operand will be the contents of the B register. In contrast, DZ indicates that the R operand will be the data from the D bus (via the shifter) and that the S operand will be zero. The complete set of ALUSOURCE functions is listed in Table 4.2.

We have now covered sufficient fields in the control word to be able to present some simple examples before moving on. Suppose we wish to add the contents of registers R5 and R8 and branch to the label @1 if the result is zero. This is accomplished with the following instruction.

```
CJP     Z,@1  AB  A=R5 B=R8  ADD
```

The ALUSOURCE field (AB) indicates that the A and B registers should be used as ALU operands, the A and B fields select registers R5 and R8, and the ALUFUN field (ADD) causes them to be added. Note the form of the A and B specifications here. As described in Chapter 2 the long form must be used since both R5 and R8 could refer to either

Table 4.2 ALU source mnemonics.

Mnemonic	ALU operands	
	R	S
AQ	A register	Q register
AB	A register	B register
ZQ	0	Q register
ZB	0	B register
ZA	0	A register
DA	D bus (via shifter)	A register
DQ	D bus (via shifter)	Q register
DZ	D bus (via shifter)	0

the A or the B field.† The condition code Z tests the result of the addition causing the CJP sequencer function to branch if it is zero. The actual result of the addition is discarded after the test; no registers are altered.

As another example, suppose we wish to test the contents of a single register, say R4, for zero. This we do as follows

```
CJP     Z,@1  ZA A=R4  OR
```

where the operands are chosen as zero and A (set to R4) and ORed together. Since ORing with zero has no effect the result will be the contents of R4, thus achieving our aim. In fact, this type of construct occurs very often and the defaults set in the standard ms initialization file allow it to be abbreviated to

```
CJP     Z,@1  R4
```

ALUSOURCE defaults to ZA, ALUFUN to OR, and a register name in isolation is taken to refer to the A register field. To avoid any confusion in later examples we will not rely on these defaults here.

The above examples both discard the ALU result. Usually, the ALU result is more valuable and must be stored. The storing of the result is controlled, at least in part, by the ALUDEST field of the control word. This field controls not only writing to the register file and the Q register, but also determines what data should be made available on the AY bus. In addition, if the result is to be stored in the register file or Q, it also allows the value to be shifted one place to the left or the right before being stored.

The most common ALUDEST function is NOP, or no operation, which causes the result to be made available on the AY bus, but otherwise to be discarded. NOP is the default value of this field and is generated by the assembler when no other value is given. This was the case in our two previous examples.

The next most common is RAMF which causes the ALU result (F) to be stored in the register file (RAM). As described above, the value is stored in the register selected by the B field. This operation has no effect on the ALU sources, and in particular, we can perform read/modify/write operations of the form

```
B := B <op> A
```

in which the same B register is used both as a source and a destination. Thus, to add registers R2 and R3, leaving the result in R3, we say

```
CONT    AB A=R2  ADD   RAMF B=R3
```

Notice the position of the B specification. Although, as has been said, the order of the fields is unimportant as far as the assembler is concerned, we position the B next to the RAMF because of the destructive nature of this function.

---

† In this case, since addition is commutative, the A and B fields could be interchanged without effect.

To move a value from one register to another without modification, say from R2 to R3 we would say

```
CONT  ZA A=R2  OR  RAMF B=R3
```

in which the value in R2 is ORed with zero before being stored in R3.

One way to set a register to zero is to exclusive OR it with itself. Whilst we could generate a zero externally to the ALU and move it into a register, this method uses fewer CPU resources, letting the rest of the CPU perform other functions.

```
CONT  AB A=R2  EXOR  RAMF  B=R2
```

An even shorter way is to AND the register with zero, as in

```
CONT  ZB  AND  RAMF B=R2
```

To increment a register by one we can add zero to it and set the carry in. The carry in is controlled by a single bit in the control word. Specifying the mnemonic CIN in an instruction causes the carry in to be set to one, the default being zero. Thus we could say

```
CONT  ZB  ADD CIN  RAMF B=R2
```

in which the B register is used as both a source and a destination, to increment register R2. It is important to note that the Am2901C inverts the sense of the carry in and out during subtraction operations. Thus specifying CIN on a subtraction causes no borrow, whereas not specifying it causes an extra one to be subtracted. Thus, to subtract R4 from R3 leaving the result in R3 we use

```
CONT  AB A=R4  SUBR CIN  RAMF B=R3
```

As with NOP, RAMF makes the result available on the AY bus possibly to be used in other parts of the CPU. In fact, the only ALUDEST function which does not do this is RAMA. RAMA is similar to RAMF, except that the output data selector makes the contents of the A register available directly on the AY bus independent of the ALU result. This allows the contents of a register to be moved to some other part of the CPU whilst an operation is being performed in the ALU. We could expand on the increment example to do this as follows.

```
CONT  ZB  ADD CIN  RAMA A=R1 B=R2
```

Here, register R2 is incremented as before, but at the same time the contents of R1 will appear on the AY bus.

To store the ALU result in the Q register we use the QREG function. Again, the result will also appear on AY.

```
CONT  AB A=R1 B=R0 AND  QREG
```

This example takes the logical AND of R1 and R0 and stores the result in Q.

The remaining four ALUDEST functions all perform a one-bit shift before storing the value. As can be seen from Figure 4.2, the shift only affects the value stored in the registers, and not that presented to the AY bus. If the shifted value is needed elsewhere in the CPU

then the value must be shifted and stored temporarily in a register to be brought out in a later instruction. Two of these functions, RAMU and RAMD, shift the result but are otherwise like RAMF. RAMU shifts up (towards the MSB) whilst RAMD shifts down (towards the LSB). The other two, RAMOU and RAMOD, shift the contents of the Q register at the same time. This allows Q to work together with any of the registers in the file to form a 64-bit, double precision register. The primary purpose of the Q register is to do just this during multiplication and division algorithms. We will investigate this in detail in the section on special functions.

Table 4.3 details the actions of all the ALUDEST functions. The final four columns in this table are concerned with what happens to the most and least significant bits during the shift operations. This will be discussed in the section on the SIN field. The AY column indicates the source of the data placed on the AY bus. These are the data tested by the OD, EV, OB, EB, OS and ES condition codes.

4.2 The shifter

The shifter is positioned between the D bus and the Am2901C array as can be seen in Figure 4.1. All data entering the ALU passes through it, allowing byte manipulation operations to take place in the same instruction as other ALU functions of the kind we have already seen. The shifter can be thought of as divided into two sections acting in series. The first section rotates the data through 0, 1, 2, or 3 bytes, whilst the second applies a mask to clear selected bytes to zero. The mask contains four bits, one for each of the bytes in the rotated value; setting a bit causes the corresponding byte to be zeroed.

In all, 64 functions are available to control the shifter, four rotations combined with 16 masks. Because of this very large number, the mnemonics have been chosen according to a formula which allows them to be reconstructed as required. Consider the value on the D bus as ABCD, where each letter corresponds to one of the bytes. A is the most significant byte and D the least. A rotation of one byte to the right (towards the least significant byte) will thus produce DABC. Similarly, rotations through two and three bytes will generate CDAB and BCDA. A byte to be zeroed is represented by the letter Z. Thus, ZBCD would perform no rotation but set the most significant byte to zero, whereas ZABC would perform a one byte rotate right first. This latter example corresponds to a one byte logical shift.

This method is used to generate all the possible shifter mnemonics. However, by far the most commonly used ones correspond to simple rotations and shifts to both the left and right. For these, an additional set of mnemonics are defined and used in preference to the corresponding general ones. These are summarized in Table 4.4. As can be seen, the special cases have somewhat greater mnemonic significance. Since the data being manipulated by the shifter is taken from the D bus we will defer presenting examples of its use until the next section, when we will have control over the source of the data.

Table 4.3 ALU destination mnemonics.

Mnemonic	RAM function		Q function		AY	RAM shift I/O		Q shift I/O	
	Shift	Load	Shift	Load		LSB	MSB	LSB	MSB
QREG	X	None	None	F -> Q	F	X	X	X	X
NOP	X	None	X	None	F	X	X	X	X
RAMA	None	F -> B	X	None	A	X	X	X	X
RAMF	None	F -> B	X	None	F	X	X	X	X
RAMU	Up	2F -> B	X	None	F	INR0	F_{31}	X	Q_{31}
RAMD	Down	F/2 -> B	X	None	F	F_0	INR_{31}	Q_0	X
RAMQU	Up	2F -> B	Up	2Q -> Q	F	INR0	F_{31}	INQ_0	Q_{31}
RAMQD	Down	F/2 -> B	Down	Q/2 -> Q	F	F_0	INR_{31}	Q_0	INQ_{31}

Note.

Refer to table 4.6 for a definition of INR_0, INR_{31}, INQ_0, and INQ_{31}.

Table 4.4 Special shifter functions.

Special mnemonic	General mnemonic	Description
RTL1	BCDA	Rotate left one byte.
RTL2	CDAB	Rotate left two bytes.
RTL3	DABC	Rotate left three bytes.
RTR1	DABC	Rotate right one byte.
RTR2	CDAB	Rotate right two bytes.
RTR3	BCDA	Rotate right three bytes.
SHR1	ZABC	Shift right one byte.
SHR2	ZZAB	Shift right two bytes.
SHR3	ZZZA	Shift right three bytes.
SHL1	BCDZ	Shift left one byte.
SHL2	CDZZ	Shift left two bytes.
SHL3	DZZZ	Shift left three bytes.
MASK	ZZZD	Select least significant byte.

4.3 The D bus field

The D bus is the main data path in the CPU (see Figure 1.1) and carries data between the cache memory, the system bus, the virtual memory translation buffer, and the ALU. We will briefly discuss it here in order to be able to present more realistic examples of the operation of the ALU and shifter. Table 4.5 lists the mnemonics for the D bus control functions together with a description of their actions.

The default value for this field is ALU. This has the effect of connecting the AY bus to the D bus, allowing the output of the Am2901C array to be recirculated through the shifter. In addition, results on AY can be sent to the cache or main memory. The recirculation is only useful when the shifter is performing a non-default function, since otherwise an operation such as

```
CONT   DZ   OR   RAMA A=R1 B=R2
```

merely results in data being moved from R1 to R2 without modification. This is much better accomplished by

```
CONT  ZA A=R1  OR  RAMF B=R2
```

which does not tie up the D bus.

When the D function is ALU, care must be taken not to create a complete, unbroken loop through the ALU. The instruction

```
CONT  DZ  ADD CIN  RAMF  B=R0
```

tries to add one to the value on the D bus and move the result into R0. However, the RAMF causes this result to appear on AY, which, because of the default ALU D bus function, is fed straight back into the D input! This naturally causes a race condition in which the actual value obtained will be undefined.

BR puts the contents of the branch field of the control word onto the D bus. In order to be able to generate both positive and negative constants, the 12-bit branch field is sign extended to 32 bits. That is, bit 11 of the branch field is copied into bits 11 to 31 of the D bus. Thus, to load the constant -76 into register R0 we would say

```
CONT  DZ D=BR,-76  OR  RAMF B=R0
```

Since D is very similar to A and B as far as the ALUSOURCE field is concerned (DZ in this example), it is conventional to use the long form for the D specification. However, the meaning would be identical if the 'D=' were to be omitted. The assembler will place a 12-bit two's complement representation of -76 in the branch field of the instruction and the sign extension will turn this into a 32-bit two's complement representation on the D bus. This allows all values in the range -2048 to 2047 to be generated directly. As the data passes though the shifter on its way into the ALU a much larger set of constants can be made by using the shifter to modify the value from the branch field.

Table 4.5 D bus control functions.

Mnemonic	Description
ALU	The output of the ALU (AY bus).
BUS	Data from the system bus.
BR	Constant from the branch field (BR).
CSH	Data from the cache.
CAIR	Read back CA, IR and I/O status.
TB	Translation buffer.
VAR	Read back virtual address register.

A common use of this occurs in bit manipulation operations. Here we often wish to set, reset, or test a single bit in a register. The sign bit, the least significant bit, and bits 26 and 27 can be treated specially by making use of the S, NS, OD, EV, OB, EB, OS, and ES condition codes which allow these bits to be tested directly. However, the best way to handle the general case is to create a mask with the required bit set to one in an otherwise zero word using the branch field together with the shifter. This can be ANDed with the value to test the bit, ORed with it to set the bit, and its complement ANDed with the value (using the NOTRS function) to reset it. To illustrate, suppose we wish to return from a subroutine if bit 19 of R4 is set. This could be done with

```
CRTN   NZ  DA D=BR,(1 << 3) SHL2 A=R4  AND
```

in which the expression (1 << 3) generates a constant with bit 3 set and the SHL2 then shifts this into bit 19. To clear the bit unconditionally consider

```
CONT   DA D=BR,(1 << 3) SHL2 A=R4  NOTRS  RAMF B=R4
```

Alternatively, because of the branch field sign extension, we can directly generate a mask with a single zero in it. ANDing with this will have the same effect.

```
CONT   DA D=BR,0FF7H RTL2 A=R4  AND  RAMF B=R4
```

Here an RTL2 is used so that the high order ones from the sign extension are rotated into the low order bits; a shift would have filled them with zeros. Note the hexadecimal value given for the branch field.

A problem arises with this kind of code when wanting to branch conditionally or call conditionally on a bit set or reset. Unfortunately we need to use the branch field twice, once to provide the mask constant, and again to give the branch address. This is, of course, impossible and the assembler will signal an error on any attempt to define a field twice in the same instruction. Usually, we have to compromise and settle for two instructions of which the first performs the masking and saves the result in a scratch register which can be tested for zero or non-zero by the second. For example

```
CONT   DA D=BR,(1 << 3) SHL2 A=R4  AND  RAMF B=R0
CJP    Z,@1  ZA A=R0  OR
```

A more subtle technique is to preserve the condition using LC, thus avoiding the need for the temporary register and freeing the second instruction to perform other functions if needed. The condition is specified and generated on the first instruction, but tested on the second.

```
CONT   Z  DA D=BR,(1 << 3) SHL2 A=R4  AND
CJP    LC,@1
```

With the D bus field taking its default value of ALU, byte manipulations on the ALU registers are easily performed. A very common example of this is to be found in the standard microcode when extracting opcodes from the code steam. Prefetched code is held in a register

called ir0 (actually R5) and whenever an opcode has been decoded it must be discarded from ir0. Code is executed starting from the least significant byte so the correct action is to shift ir0 right one byte. This we do with

```
CONT  DZ SHR1  OR  RAMA A=ir0 B=ir0
```

which puts the initial contents of ir0 onto the AY bus and in turn onto the D bus, takes the D bus value and shifts it right one byte, and finally stores the shifted value back into ir0. In the real case, as we will see later, this instruction usually performs other functions at the same time.

As a final example we will consider the problem of inserting a byte into a 32-bit value held in a register. We will take the byte from the branch field and insert it into byte one (next to least significant byte) of R3.

```
CONT  DZ ABZD  OR  RAMA A=R3 B=R3
CONT  DA D=BR,<exp> ZZDZ A=R3  OR  RAMF B=R3
```

The first instruction sets byte one of R3 to zero so that the value of <exp> can be ORed in on the second, in which the value from the branch field is moved into byte one with the ZZDZ shifter function.

The remaining five D bus functions are concerned with the system bus interface, the cache memory, and the virtual memory management unit. We defer consideration of these functions until the chapters covering those areas of the CPU.

4.4 The SIN field

When considering the ALUDEST field in an earlier section the question of what happens to the bits shifted into or out of the most or least significant bits on the RAMU, RAMD, RAMQU, and RAMQD operations was left unanswered. This section attempts to provide the answer. However, as most of the SIN functions are used only rarely and many have quite subtle effects, this section may be skipped at first reading.

There are two distinct sets of behaviour for the SIN (serial in, pronounced S-in) functions, referred to as the 'normal set' and the 'alternate set'. Usually, an SIN function is specified because a simple shift is being performed, in which case the normal set is used and the microprogrammer has complete control over the bit shifted in. The alternate set is selected automatically when certain of the special functions are used (see the next section), or when, on the previous cycle, the programmer has explicitly used the special function ASIN (alternative SIN).

An understanding of this behaviour is important because although a different set of four mnemonics is provided for the alternate functions, there are only two bits in the control word for SIN. These two bits are interpreted differently by the hardware when some condition (not encoded in the SIN field itself) selects the alternate set. The alternate mnemonics are provided simply to enhance the readability of the code and specifying an SIN mnemonic from the alternate set will not on its own cause the alternate functions to be used.

Figure 4.3 Normal SIN functions.

ALUDEST	SIN field, normal set mnemonics							
	ZERO		ONE		ROT		ARI	
	R	Q	R	Q	R	Q	R	Q
RAMU	← 0	X	← 1	X	←	X	← 0	X
RAMD	0 →	X	1 →	X	→	X	S →	X
RAMQU	← ← 0		← ← 1		←	←	← ← 0	
RAMQD	0 → →		1 → →		→	→	S → →	

Notes.

S The ALU sign bit (sign extension).
X Preserved.

Figure 4.4 Alternate SIN functions.

ALUDEST	SIN field, alternate set mnemonics							
	US		DROT		QRSAVE		TC	
	R	Q	R	Q	R	Q	R	Q
RAMU	← 0	X	← 1	X	←	X	← 0	X
RAMD	C →	X	Q_0 →	X	B →	X	CS →	X
RAMQU	← ← D		← ←		←	← 0	← ← D	
RAMQD	C → →		→ →		B → →		CS → →	

Notes.

C The ALU carry out.
Q_0 The least significant bit of the Q register.
B^0 The bit saved by QSAVE or RSAVE on the previous cycle.
CS The corrected sign bit.
D One bit in the result of a division step.
X Preserved.

Figure 4.3 attempts to illustrate diagrammatically the behaviour of the normal SIN functions whilst Figure 4.4 does the same for the alternate set. Table 4.6, which should be used in conjunction with Table 4.3, specifies the actions more formally.

We will begin by considering simple cases of the use of the normal set. The single precision ALUDEST functions RAMU and RAMD shift only the value which will be stored in the selected B register. The Q register is not affected. The default SIN function is ZERO which results in a logical shift in which the vacated bit position is filled with zero. It is normal to specify the function explicitly to make this clear. Thus to shift R7 right one place we could say

```
CONT  ZB  OR  RAMD ZERO B=R7
```

or to shift it one place left

```
CONT  ZB  OR  RAMU ZERO B=R7
```

To shift it left and set the new LSB we would use the ONE function, as in

```
CONT  ZB  OR  RAMU ONE B=R7
```

ONE can also be used to set the MSB in a right shift.

A trick which is sometimes useful is to perform a shift of two places to the left in a single operation by first adding a register to itself. In this case we can control the state of both the vacated bits since we can optionally inject a carry into the addition. Thus to shift left two setting bit 1 and clearing bit 0, consider

```
CONT  AB A=R7  ADD CIN  RAMU ZERO B=R7
```

The ROT function results in the filling of the vacated bit by the bit shifted out from the opposite end of the register, whilst the ARI function performs an arithmetic shift. For a shift left this is the same as ZERO, but for a shift right the MSB is filled with a copy of its previous value. That is, sign extension is performed. Thus if the value in R7 is considered as a two's complement number and is negative, then the value will still be negative after

```
CONT  ZB  OR  RAMD ARI B=R7
```

Note however, that even an arithmetic shift right is not equivalent to division by two for two's complement numbers.

The double precision shift functions RAMOU and RAMOD behave just like RAMU and RAMD except that the value in Q is shifted at the same time. The functions ZERO, ONE, and ARI all treat the selected B register (which we will refer to as R) and Q together as a 64-bit register in which R holds the most significant word, and Q the least significant word. Thus, on a right shift with these functions, the LSB of R moves into the MSB of Q, whilst on a left shift, the MSB of Q moves into the LSB of R. In other respects these functions operate like the corresponding single precision ones, with the bit being shifted in on a shift left going into the LSB of Q. Thus shifting R7 and Q left together bringing in zero is done with

```
CONT  ZB  OR  RAMOU ZERO B=R7
```

In contrast, the ROT function treats R and Q as two independent registers which are both rotated. The effect on R is identical to the single precision case. Rotating the combination as a single 64-bit

Table 4.6 Actions of the SIN functions.

SIN function	RAMU	RAMD	RAMQU		RAMQD	
	INR_0	INR_{31}	INR_0	INQ_0	INR_{31}	INQ_{31}
ZERO	0	0	Q_{31}	0	0	R_0
ONE	1	1	Q_{31}	1	1	R_0
ROT	R_{31}	R_0	R_{31}	Q_{31}	R_0	Q_0
ARI	0	S	Q_{31}	0	S	R_0
US	0	C	Q_{31}	D	C	R_0
DROT	1	Q_0	Q_{31}	R_{31}	Q_0	R_0
QRSAVE	R_{31}	B	R_{31}	0	B	R_0
TC	0	CS	Q_{31}	D	CS	R_0

Notes.

C	The ALU carry out.
S	The ALU sign bit (sign extension).
B	The bit saved by QSAVE or RSAVE on the previous cycle.
CS	The corrected sign bit.
D	One bit in the result of a division step.
Q_0	The least significant bit of the Q register.
Q_{31}	The most significant bit of the Q register.
R_0	The least significant bit of the selected B register.
R_{31}	The most significant bit of the selected B register.

register can be done with the alternate DROT function.

The alternate SIN functions are only available in conjunction with the special function field. The description is therefore deferred until the next section.

4.5 The special functions

The special function field is used to control many rarely used, but nevertheless important features of the CPU. Since these functions are all encoded in a single field, only one can be specified on a single instruction even though apparently unrelated parts of the CPU are involved. Further, the special function field is itself used for another purpose, namely, to control the instruction register (IR) and the cache address register (CA). We will not mention this further here except to

say that a special function cannot be specified on any instruction which manipulates IR or CA. Since the special functions, as their name suggests, perform very specialized operations, these restrictions are seldom a problem. As with the previous section, this one may be skipped at first reading.

The special functions are summarized in Table 4.7. We will discuss in detail here only those which are concerned with the operation of the ALU. The rest will be covered in the chapters covering the related parts of the CPU.

Referring to the table, the first six entries are related to multiplication and division. Multiplication is performed using the shift and add algorithm, and division is done using the non-restoring algorithm. In

Table 4.7 The special functions.

Mnemonic	Description
MUL	Unsigned or two's complement multiplication step.
USDIVF	Unsigned division first step.
TCDIVF	Two's complement division first step.
TCDIV	Two's complement division intermediate step.
USDIV	Unsigned division intermediate step.
DIVL	Unsigned or two's complement division last step.
ASIN	Select alternative SIN on next cycle.
CSAVE	Save the ALU carry out (C).
RSAVE	Save bit shifted from R_0 on RAMD or RAMQD.
QSAVE	Save bit shifted from Q_0 on RAMQD.
OUTINT	Interrupt diagnostic Z80 for output.
ININT	Interrupt diagnostic Z80 for input.
WRMM	Write to memory management mode table.
TBWR	Write to translation buffer.
CLRPERR	Clear cache, tr buf and main memory parity errors.

the case of two's complement division, the algorithm works directly on numbers in the two's complement representation, avoiding the overhead of conversion to sign and magnitude. Whilst it would be possible to do multiplication and division without the aid of these special functions it would be significantly slower since, because of the instruction pipeline, two or three cycles would be needed for each bit. These functions work by modifying the ALUSOUCE and ALUFUN fields in the instruction and allow the operation to take place at one cycle per bit. At the same time as modifying the ALU fields, the SIN field is switched to the alternate set allowing bits in the answer, computed by the special function logic, to be shifted in as the operation proceeds. The SIN function must be specified as TC or US according to whether the numbers are regarded as two's complement or unsigned.

These functions are only of use for these specific operations, and so instead of attempting to give a precise definition of all their actions, we refer the interested reader to Chapter 10 and Appendix E, in which examples of their use are presented.

The ASIN function allows the programmer direct access to the alternate SIN functions. However, because this is part of the special function logic there is a one cycle delay in the operation of this function. Referring to Figure 4.4, the only single precision function which is of any significant use (except in the context of multiplication and division) and which cannot be obtained in the normal set is QRSAVE. This function causes the MSB of the selected register to be set to a value which was saved on the previous cycle by a QSAVE or RSAVE special function (see below). If the previous cycle did not specify one of these, the value obtained is undefined.

RSAVE should only be used on an instruction containing a RAMD or RAMQD ALUDEST function. It causes the bit shifted out of the LSB of the register to be saved in a special place from where it can be retrieved by a QRSAVE on the next cycle. The bit must be used immediately as it will only be stored for one cycle; furthermore, the SIN function will be switched to the alternate set on the next cycle to make the QRSAVE possible. QSAVE is similar except that the saved bit is taken from the LSB of Q. These functions facilitate shifting when multiple precision values are held in several registers, as for example in floating point arithmetic routines. For example, to perform an arithmetic shift right a 96-bit value held in R2 (most significant), R1, and R0 (least significant) we would write

```
CONT  ZB  OR  RAMD ARI B=R2  RSAVE
CONT  ZB  OR  RAMD QRSAVE B=R1  RSAVE
CONT  ZB  OR  RAMD QRSAVE B=R0
```

This is significantly faster than could be achieved by testing bits and branching.

For the double precision case, in addition to QRSAVE the DROT function is of value.

```
CONT  ASIN
CONT  ZB  OR  RAMQU DROT B=R0
```

is the 64-bit equivalent of

```
CONT  ZB  OR  RAMU ROT B=R0
```

and similarly for RAMQD.

The CSAVE function causes the carry out of the ALU (C) to be stored and injected into the carry in on the next cycle. This facilitates multiple precision arithmetic. Thus to subtract a 64-bit number in R3 R2 from another in R1 R0, leaving the result in R1 R0 we can do

```
CONT  AB A=R2 B=R0  SUBR  RAMF  CSAVE
CONT  AB A=R3 B=R1  SUBR  RAMF
```

The CSAVE function can also be used to perform a multiple precision left shift by using addition to perform a one bit shift. To shift the 96-bit value held in R2 R1 R0 left again, bringing in a 1 to the LSB we would write

```
CONT  AB A=R2  ADD CIN  RAMF B=R2  CSAVE
CONT  AB A=R1  ADD  RAMF B=R1  CSAVE
CONT  AB A=R0  ADD  RAMF B=R0
```

It is important not to specify CIN on the cycle following a CSAVE as a carry would then be forced regardless of the saved value.

The remaining special functions are concerned with parts of the CPU other than the ALU and will be discussed in later chapters.

CHAPTER 5

The Map Tables and Instruction Register

The map tables provide a general purpose decoding facility which is most often employed to decode abstract machine opcodes. When an opcode has been fetched, a decision must be taken as to what action to perform in order to execute the opcode. The map tables in effect provide a hardware implementation of the case selection statement of many high level languages. In this instance the opcode is used as the case variable and the cases correspond to the actions of the opcodes.

To illustrate this, consider the BCPL example in Figure 5.1 which implements the core of an extremely simple interpreter.

Figure 5.1 A simple interpreter

```
let interp () be
$(
        switchon nextop () into
        $(
                case OP1:
                        doop1 ()
                endcase

                case OP2:
                        doop2 ()
                endcase

                case OP3:
                        doop3 ()
                endcase

                default:
                        badopcode ()
                endcase
        $)
$) repeat
```

The skeleton of a microcode implementation of this is shown in Figure 5.2. The actions of the BCPL procedures doop1, doop2, and doop3 are assumed to have been expanded into inline microcode following the entry points. The last action of each of these entries is to branch to the label nextop, where there is microcode to get the next opcode and load it into the instruction register. Once the instruction register has been loaded the CJV transfers control directly to the entry for the new opcode. In a real case the code at nextop may also be expanded inline for each of the entries to allow greater overlapping of independent functions and a considerable reduction in the number of microinstructions executed.

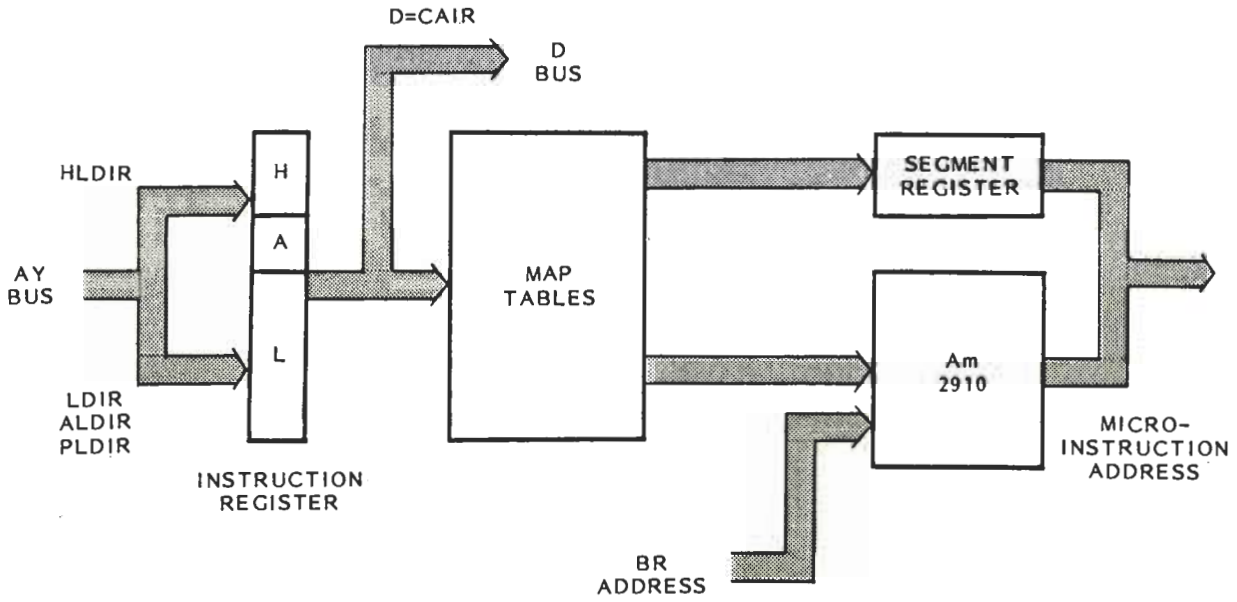
Figure 5.2 An interpreter in microcode.

```
ENTRY interp + op1
        /* Perform op1 here */
        .
        .
        JUMP   nextop
        CONT
/*--------*/

ENTRY interp + op2
        /* Perform op2 here */
        .
        .
        JUMP   nextop
        CONT
/*--------*/

ENTRY interp + op3
        /* Perform op3 here */
        .
        .
        JUMP   nextop
        CONT
/*--------*/

DEFAULTENTRY        interp
        JUMP   trap
        CONT
/*--------*/

nextop:
        /* Get next opcode into instruction register */
        .
        .
        CJV
        CONT
/*--------*/
```

As the chapter proceeds, the operation of this interpreter should become clear. Although this example has been grossly simplified it does represent a miniature version of the standard instruction set.

Figure 5.3 shows the relationship of the instruction register (IR), map tables and sequencer. The IR is loaded with the item to be decoded, where it provides an address into a map table. The contents of the selected map table entry are made available to the sequencer which, by means of the CJV function, treats it as a control store address. Control is then passed to that address, where code will be found to perform the actions required for the decoded value. This operation is illustrated schematically in Figure 5.4.

Figure 5.3 The map tables.

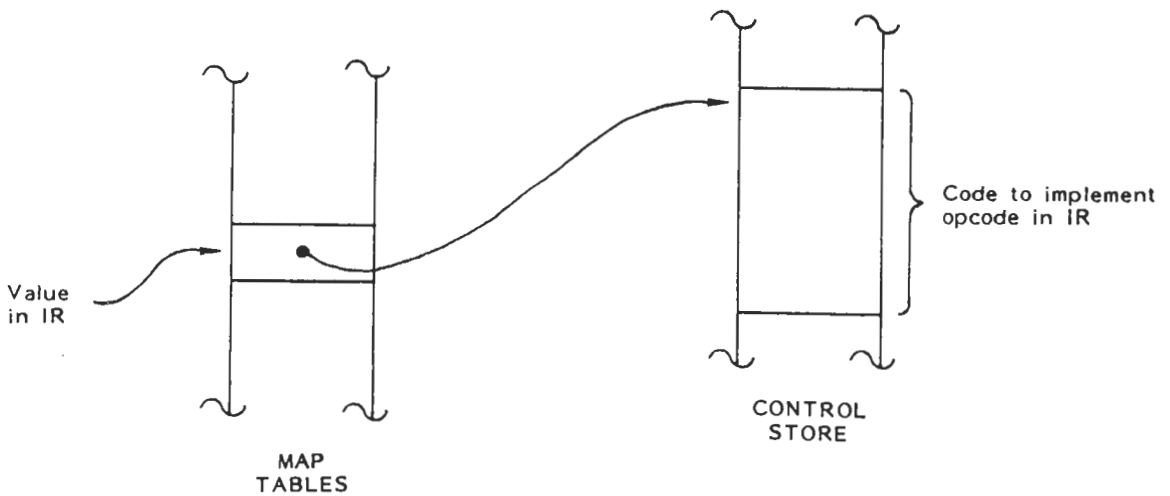


Although the map tables are intended primarily for opcode decoding, they can equally well be used for any similar decoding function, for example, decoding the addressing modes used by conventional register machines.

5.1 The map tables

Each map table contains 256 entries, the optimum number assuming that the items to be decoded are bytes. This will often be the case if the machine is running a conventional language with a linear instruction

Figure 5.4 Action of decoding.



stream. The standard instruction set is of this type. However, the decoding mechanism is just as useful whatever the representation of the program.

In order to allow for more than 256 opcodes, the map tables are paired. The second member of each pair is typically used for the less frequently encountered opcodes. These might be encoded in two bytes with the first 'escape' byte indicating that the second byte should be decoded via the alternate table. An instruction set is thus considered to occupy a pair of map tables.

So that several entirely independent instruction sets may reside in the machine simultaneously, several map table pairs are provided. The architecture allows for up to 16 such pairs, but current implementations have only four.† A four bit register holds the current instruction set number. Typically this would be loaded by operating system software when a process is dispatched, but there are interesting possibilities for inter-language procedure calls.

5.2 The instruction register

Referring back to Figure 5.3 we see that the instruction register is divided into three sections labelled H, A, and L. The L section is eight bits wide and is used to select one entry from the 256 entries in a single map table. This is by far the most frequently used section of the instruction register and often the term 'instruction register' is used loosely to refer to just this section. The A section is a single bit which usually contains zero. It can be set to one to allow access to the alternate 256 entries in the second member of a map table pair. For the time being we will ignore the H section.

The instruction register is controlled by a shared field in the control word. This is the same field as the one which controls the cache address register and the special functions. As a result of this the instruction register cannot be manipulated on a microinstruction that specifies a special function and certain obscure combinations of cache address and instruction register functions are unavailable. Table 6.2 in the next chapter fully covers this restriction.

As with all fields, the default IR function (NOPIR) results in no modification to the contents of the IR. Thus it will retain its contents until a microinstruction explicitly modifies it. The most common function is LDIR, or load instruction register, which loads the L section with the low order eight bits of the AY bus. At the same time the A section is cleared to zero. In the standard instruction set code is fetched from memory eight bytes at a time and cached in two of the Am2901C registers which we call ir0 and ir1. ir0 gets the low order four bytes, the four to be executed first, and these are transferred one by one into the instruction register prior to execution. This is usually done with

---

† Field upgradable to 16 when the next generation of memory devices is available.

```
        CONT   DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
```

which puts the contents of ir0 onto the AY bus for loading into the IR and at the same time shifts the remaining code right by one byte to discard the new opcode now being decoded. (Remember that the default D function connects AY back to D.) Any time after the LDIR, the contents of the selected map table entry are available to the sequencer for use by a CJV. Thus, within the framework of the standard instruction set, the minimum action which must be performed by an opcode is to reload the instruction register and dispatch the next instruction, as in

```
ENTRY noop
        CONT   DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
        CJV
        CONT
```

From the point of view of the high level abstract machine this is a no-op. Usually, these functions will overlapped with the active part of a real instruction so that the decoding of the next instruction is carried out in parallel with the execution of the current one.

In the standard instruction set several of the opcodes have arguments encoded as a byte following the opcode in the code stream. The high level assembler arranges that the opcode and argument do not cross a word boundary so that the argument will always be found as the least significant byte of ir0. This is because the current opcode will have been removed from ir0 after being loaded into IR by the previous opcode. In this case, the argument byte must be extracted from the code stream to expose the next opcode. An IR function, PLDIR, is provided to facilitate this. PLDIR is similar to LDIR but loads the instruction register from bits 8 to 15 of AY instead of bits 0 to 7. Thus the argument can be picked up from byte 0 of ir0 using the ZZZD shifter function to mask it off, whilst IR is reloaded from byte 1. After this, two bytes must be discarded from ir0.

```
        CONT   DZ ZZZD  OR  RAMA A=ir0 B=R0  PLDIR
        CJV    DZ SHR2  OR  RAMA A=ir0 B=ir0
        CONT
```

When it is required to decode a byte using the alternate member of a map table pair, the function ALDIR is used. This is the same as LDIR except that the A section of the instruction register is set to one when the L section is loaded. As has been mentioned, one way of using this function to extend an instruction set to 512 opcodes is to have a one byte 'escape' opcode which simply causes the following byte in the code stream to be loaded into IR with an ALDIR function. The following byte will therefore be decoded using the alternate table. The microcode to implement the escape opcode is very similar to the no-op above

```
ENTRY escape
        CONT   DZ SHR1  OR  RAMA A=ir0 B=ir0  ALDIR
        CJV
        CONT
```

In the standard instruction set ir0 will be completely filled with zeros after four bytes of code have been executed. This situation is used to make code fetching transparent to the individual opcodes. At some stage the zero in the least significant byte of ir0 will be loaded into the instruction register, selecting the zeroth entry in the map table. This entry is reserved to point to microcode which refills ir0. Depending on the value in the abstract machine program counter (which is also held in one of the Am2901C registers), either the contents of ir1 are moved into ir0, or eight more bytes of code are fetched from memory and stored in ir0 and ir1.

This mechanism allows the assembler to guarantee that opcodes with byte arguments do not cross word boundaries. If only one byte is left in a word when a two byte opcode is needed, that byte is filled with zero. The zero byte is not a no-op; it actually causes the automatic fetching mechanism to be invoked one byte earlier than would otherwise have been the case.

When any form of control transfer takes place in the abstract machine prefetched code in ir0 and ir1 must be discarded and replaced with code from the destination address. To assist with this an IR function called FETCH is provided.

```
CONT  FETCH
```

is entirely equivalent to loading IR with 1, as in

```
CONT  DZ D=BR,1  OR  LDIR
```

It is provided so that this can be done without the use of other CPU resources. In the standard instruction set entry 1 points to microcode which goes directly to memory to fetch new code, discarding any that has been prefetched.

The final IR function is HLDIR. This loads the H section of IR from bits 0 to 3 of AY. This function is used to select an instruction set by using the instruction set number as the high order part of the map table address. Normally this function will be used very rarely, probably in context switching microcode which needs to select an appropriate instruction set for the process which is about to run.

Referring back to the example of the simple interpreter of Figure 5.2, each of the entry points defines a map table entry offset by the opcode value from the base interp. interp is a constant which defines which map table is to be used. Thus to put these opcodes into the normal set of instruction set 1, interp should be defined by

```
interp = 1 << 9
```

which sets the value 1 in the H section of the map table address. Note also the DEFAULTENTRY statement. It is most important that all entries in a map table be defined. Since the instruction register is loaded with byte values from the instruction stream, an errant program could provide an illegal value. If the map table entry corresponding to that value were undefined then either a map table parity error would be detected, or control would be passed to a random microcode store address where a control store parity error would probably occur. In any event, the operation of the machine would be undefined and a fatal

system crash would follow.

The DEFAULTENTRY statement indicates to the microlinker that any unused entries in the map table whose base address is given as the argument to the DEFAULTENTRY, should be defined to point to the following instruction. In this case a branch is performed to the label trap, where, presumably, appropriate action occurs. This simplifies maintenance, since the linker will automatically compensate if opcodes are added or removed.

Table 5.1 summarizes the effects of the IR functions on the three sections of the instruction register.

It is useful to be able to determine the contents of IR at certain times. This is particularly useful for opcode profiling, when special microcode dynamically tracks the frequency of occurrence of each opcode during execution. This subject is discussed in Chapter 10. When several similar opcodes must be implemented they can sometimes share microcode, and reading the IR allows the microcode to determine which particular opcode is being executed.† The IR is read onto the D bus with the D=CAIR function. As its name suggests, this function also reads back the cache address register at the same time (see the next chapter) and certain other information. When this function is specified, the H, A and L sections of IR are placed, as a single 13-bit number, on the low 13 bits of D. It is permissible to reload the instruction register on the same cycle as reading it. Since the load does not actually occur until the very end of the cycle, the value read will be the old contents.

The L section of IR also has another role when communicating with the diagnostic microprocessor (DP). It is used as a bi-directional 8-bit data port. This is a rather specialized function intended mainly for

Table 5.1 IR function mnemonics.

Mnemonic	Action		
	H	A	L
NOPIR	Hold	Hold	Hold
LDIR	Hold	0	Load from AY0-AY7
PLDIR	Hold	0	Load from AY8-AY15
ALDIR	Hold	1	Load from AY0-AY7
FETCH	Hold	0	1
HLDIR	Load from AY0-AY3	Hold	Hold

† This is a computed come-from operation.

diagnostic use and for dynamic loading and unloading of the control store and map tables. We defer discussion of this until Chapter 9.

CHAPTER 6

The Cache Memory

The cache memory provides a large bank of fast registers internal to the CPU. At the microcode level the cache is simply a randomly addressable memory, separate from the main system memory. In implementing computer languages it is often possible to identify areas of locality of reference to data; this knowledge can be used to ensure that such data are likely to be present in the cache memory when required. Although this will be achieved in different ways for different languages, it can be accommodated in ORION. The various instruction sets can use the cache memory, which is under the full control of the microprogrammer, in different ways. This should be contrasted with a cache memory in a conventional minicomputer which is designed to speed access to frequently used but arbitrarily distributed memory locations.

In the standard system, a section of the cache memory is used to hold the top of the scalar stack where local storage and expression evaluation take place in the C language, whilst another section contains memory management information used by the operating system. When used like this the cache is just as effective at reducing the number of memory references as in a conventional system, and in addition flexibility is retained for other applications.

6.1 The cache data path

The data path to the cache is attached to the D bus (see Figure 1.1). As we will see in the next chapter, this allows data to be moved between the cache and main memory at the maximum rate of the main memory.

Data is read from the cache with the D=CSH D bus function. For example, the contents of the currently addressed cache location could be moved into R4 with

```
CONT  DZ D=CSH  OR  RAMF B=R4
```

Writing is controlled by an independent one-bit field which is set by specifying the mnemonic CWR in a microinstruction. This causes the data on the D bus to be written to the currently addressed cache location. Thus, R4 could be moved back into the cache with

```
CONT  ZA A=R4  OR  CWR
```

To improve the performance, there is a one level pipeline in the read path (but not in the write path). This means that an extra cycle is needed after a change of the addressed location before the new data becomes available (during which the old data is still available), but thereafter in a block transfer, one word can be read per cycle (see next section). Note, however, that after a CWR operation the newly written data can be read back on the following cycle.

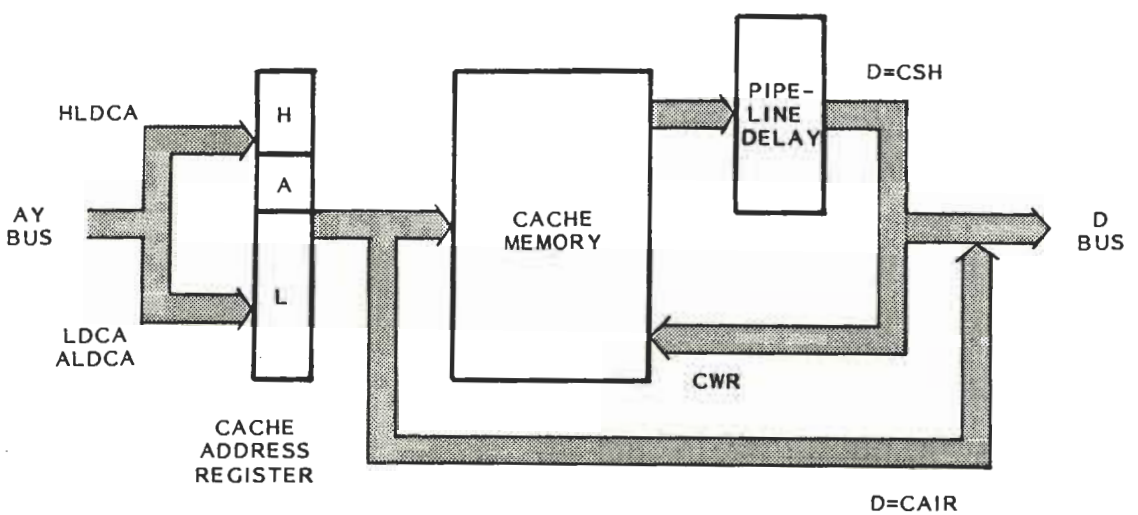
6.2 The cache address register

The cache is addressed by a versatile register which supports the segmentation and the push and pop operations needed by a stack. This register, known as the cache address register (CA), is similar to the instruction register. It is illustrated in Figure 6.1. CA is divided into three sections, H, A, and L. The L section is nine bits wide and implemented as a loadable up/down counter. Thus it can be loaded, incremented, or decremented. The A section is a single bit allowing easy selection between two 512 word sections of the cache, whilst the H section is four bits wide and selects one of up to 16 such pairs. However, current implementations of the machine have only two pairs.† The A bit has been separated from H so that its state can be controlled directly when the L section is loaded. This gives slightly greater flexibility when more than one 512 word section is in use. The H section can be loaded independent of A and L and may be used in much the same way as the H section of IR. For example, different cache banks can be dedicated to different instruction sets or different processes to reduce the amount of data movement between cache and main memory on context switches.

The default CA function (NOPCA) results in no change to any section of CA. This will be generated automatically by the assembler if no other CA function is specified.

In the standard instruction set the normal half (A = 0) of one cache bank is used as a stack top cache. The L section of CA is a copy of the low nine bits of a stack pointer (sp) held in one of the Am2901C registers. The L section is loaded with the LDCA function. Thus, to establish the copy of the stack pointer in L we could say

Figure 6.1 The cache memory.



† Field upgradable to 8 when the next generation of memory devices is available.

```
CONT  ZA A=sp  OR  LDCA
```

which moves the value into L, clears A to zero, and does not affect H. A similar function allows a location in the alternate half of the same bank to be accessed. This function is ALDCA which is the same as LDCA except that A is set to 1. To access location 23 in the alternate half we do

```
CONT  DZ D=BR,23  OR  ALDCA
```

In the standard system the alternate half of one cache bank is used to store a set of 'privileged registers' used by the operating system to control memory management and context switching.

The value in the L section of the cache address register can be incremented or decremented without affecting either the A or the H section. These operations can be performed by specifying the function INCCA or DECCA in a microinstruction. For the purposes of incrementing and decrementing, the L section is treated in isolation as a nine-bit up/down counter. There is no indication of overflow or underflow; the counter simply wraps round. This behaviour is quite deliberate because it allows a section of the cache to be treated as a circular buffer holding a region at the top of the stack. The microprogrammer has the responsibility of ensuring that data is moved between cache and main memory as required to prevent overflow. This is done in the standard system by using an Am2901C register as a cache base register (cb) to point to the lowest address assumed to be held in the cache. When a procedure call opcode is executed a check is made by the microcode to see if there is enough 'headroom' in the cache. If not, data is moved into memory two words at a time starting from the bottom until there is sufficient room, with the cb register being incremented as appropriate. Once the data has been copied into memory no further action is needed. The vacated space can now be re-used at the top of the stack because of the wrap around feature. A similar process occurs in reverse if a procedure return opcode discovers that the cache is almost empty. In practice, the cache segments are sufficiently large that very little swapping is done with typical programs.

As has been mentioned there is a one cycle delay before new data can be read after a change of CA. This is illustrated by the following example.

```
CONT  DZ D=23  OR  LDCA
CONT  DZ D=CSH  OR  RAMF B=R0
CONT  DZ D=CSH  OR  RAMF B=R1
```

The first instruction loads the L section of CA with 23. The loading actually occurs at the end of the cycle. The second reads the cache data and stores it in R0. Because of the pipeline delay the data read will be the contents of the location addressed by CA before is was changed. The final instruction reads the cache data again, storing it in R1. This time the new data is read, that is, the contents of location 23.

To avoid delays caused by this pipeline a convention has been adopted in the standard instruction set whereby CA is left pointing at the top item on the stack at the end of any high level instruction.

Furthermore, CA is never changed on the last cycle of an instruction. This allows the top item on the stack to be accessed directly on the first cycle of the following instruction if needed. As an example, the following implements a stack machine 'plus' opcode which removes the top two stack items and replaces them by their sum. This is similar to the integer add instruction in the standard C instruction set.

```
ENTRY plus
        CONT   DZ D=CSH  OR   RAMF B=R0  DECCA
        CONT   DZ SHR1   OR   RAMA A=ir0 B=ir0  LDIR
        CJV    DA D=CSH A=R0   ADD   RAMF B=R0
        CONT   ZB   SUBR   RAMA A=R0 B=sp   CWR
```

The first cycle reads the cache (the top of stack) and stores the value in register R0. The DECCA function causes the L section of CA to be decremented at the end of the first cycle in preparation to read the second operand. Because of the pipeline delay, the second operand cannot be read until the third cycle but the second cycle is not wasted since the instruction register can be reloaded. This must be done before the CJV on the next to last cycle. The second cycle therefore reloads IR from the low byte of register ir0 and shifts ir0 right by one byte. The third cycle reads the second operand and adds it to the first in R0. Finally the result is written back on the last cycle. Since two items have been removed from the stack and only one has been put back the DECCA has left CA pointing to the new top of stack. However, the sp register in the Am2901Cs which holds the full 32-bit machine stack pointer must be decremented to keep it in step with CA. This is done on the last cycle whilst the result is being put back using the RAMA ALUDEST function. This whole sequence of four microinstructions takes only slightly more than half a microsecond to execute, after which the very next microinstruction executed will be the first of the newly decoded opcode.

To illustrate INCCA we will take another example based on the standard instruction set. This is the ln1 instruction which loads the value 1 onto the top of the stack. (In the C language the boolean value true is represented by one so this is quite a common operation.)

```
ENTRY ln1
        CONT   DZ SHR1   OR   RAMA A=ir0 B=ir0  LDIR
        CJV    ZB   ADD CIN  RAMF B=sp   INCCA
        CONT   D=BR,1   CWR
```

The first instruction simply reloads the instruction register as in the previous example. This must be done first so it happens before the CJV. The second cycle increments sp and CA to point to the location to be filled with the value 1. Finally a one is generated on the D bus from the branch field and written to the cache. Because there is no pipeline delay in the write path the data will be written during the third cycle to the newly addressed location. Further, that same value will be available on the next cycle should the next opcode begin with a microinstruction containing a D=CSH function.

The H section of CA is loaded independently of the A and L sections using the HLDCA function. This loads the value from bits 0 to 3 of AY into H. Normally this function will be used rarely, probably in

Table 6.1 CA function mnemonics.

Mnemonic	Action		
	H	A	L
NOPCA	Hold	Hold	Hold
LDCA	Hold	0	Load from AY0-AY8
ALDCA	Hold	1	Load from AY0-AY8
INCCA	Hold	Hold	Increment mod 512
DECCA	Hold	Hold	Decrement mod 512
HLDIR	Load from AY0-AY3	Hold	Hold

context switching microcode which needs to select an appropriate cache bank for the process which is about to run.

Table 6.1 summarizes the effects of the CA functions on the three sections of the cache address register.

It is sometimes useful to be able to determine the contents of the CA. This is less frequently the case than with IR, at least in the standard system. The CA is read onto the D bus with the D=CAIR function and when this function is specified, the H, A, and L sections of CA are placed, as a single 14-bit number, on bits 16 to 29 of the D bus. It is permissible to reload, increment, or decrement the CA on the same cycle as reading it. Since the change does not occur until the very end of the cycle, the value read will be the old contents.

As was mentioned in the previous chapter the CA is controlled by the same field in the control word as the IR and the special functions. As a result of this the cache address cannot be manipulated on a microinstruction that specifies a special function, and certain obscure combinations of cache address and instruction register functions are unavailable. Table 6.2 gives a summary of these restrictions.

Table 6.2 CA/IR restrictions.

CA/IR	NOPIR	LDIR	PLDIR	ALDIR	FETCH	HLDIR
NOPCA	yes	yes	yes	yes	yes	yes
INCCA	yes	yes	yes	no	no	no
DECCA	yes	yes	yes	no	no	no
LDCA	yes	no	no	no	yes	no
ALDCA	yes	no	no	no	no	no
HLDCA	yes	no	no	no	no	no

Notes.

Yes indicates the combination is allowed.
No indicates the combination is not allowed.

CHAPTER 7

Physical Memory and the System Bus

The physical memory and the I/O subsystems are accessed by the CPU via the system bus. In this chapter we will cover the operation of the bus and the memory modules from the point of view of the microprogrammer. Since I/O devices are accessed in the same address space as the physical memory there is little difference between device registers and memory locations (save that some are read only, some are write only, and many are less than 32-bits wide). In the examples which follow we therefore concentrate exclusively on memory operations.

7.1 Physical memory

Physical addresses are 26 bits wide and address words, allowing a total physical address space of 256 Mbytes (64 Mwords). This address space is divided into 1/2 Mbyte logical 'slots'. Each memory module or I/O subsystem occupies one or more of these logical slots. By convention memory resides in the low half of the address space (address bit 25 = 0) and I/O systems in the upper half (address bit 25 = 1).

The memory modules themselves have two way interleaving. This means that in each module two independent banks of memory operate in parallel so that in a single memory operation two words of data can be transferred in only one more bus cycle than would be required for a single word. This feature can be exploited directly by the microprogrammer to increase throughput when blocks of data need to be moved. For example, the standard instruction set fetches code from memory eight bytes (two words) at a time thus reducing the overhead of instruction fetching. If two words are to be accessed in one operation then the address sent must be even. The two words accessed will then be the one at the addressed location together with the one at the next higher (odd) address. For historical reasons the even word is referred to as the 'left word' and the odd word as the 'right word' in each pair.

Parity error protection (one bit per word) is provided by the memory modules. Parity is generated by the CPU (or a DMA device) during a write operation, transmitted over the bus, and stored along with the data. During a read operation, the parity is checked by the CPU and if an error condition is detected this will be stored. The microcode must test for stored error conditions using the PE condition code at convenient points. This is typically done in high level instruction fetch microcode.

7.2 The system bus

The system bus consists of 32 bi-directional data/address lines, a clock line, and a number of control lines. The bus, which supports multiple direct memory access (DMA) channels, is synchronous, which means that all activity on the bus is timed by a single clock signal. This bus

clock signal is generated by the CPU and its period varies along with the variable CPU clock, keeping the bus in synchronism with the CPU. The bus can operate at up to the maximum rate of the CPU (8MHz) giving it a throughput of 32 Mbytes/sec. However, since addresses and data must be transmitted over the same lines the maximum data throughput is lower; approximately 16 Mbytes/sec when reading from memory and 21 Mbytes/sec when writing.

In order to achieve the very high speed operation of the bus a three bit function code is transmitted on three of the control lines of the bus by the current bus controller during each clock cycle. This code specifies the action that will be performed during the following cycle. We have here another example of a one level pipeline; this one allows the decoding of the control code to be overlapped with the execution of the previous one thereby increasing the maximum rate of the bus. This control code is decoded by each device on the bus in order to decide when to become active and when to transmit data on the bus.

7.3 The system bus interface

As with other sections of the CPU the bus interface is under the direct control of the microcode. When the CPU is in control of the bus the function code is derived from a three bit field in the control word. However, the code specified by the microprogrammer may be modified by the hardware before actually being transmitted on the bus. This allows the memory management unit (MMU) to enforce protection if invalid memory accesses are attempted. We defer consideration of this until the next chapter and assume in the examples here that all memory operations are considered 'valid' by the MMU.

A memory access proceeds by a number of steps spread over several cycles of the CPU. On each cycle the control word must specify the operation to be performed. Although this means that the microprogrammer must be more familiar with the operation of the bus than if memory transactions were completed automatically by the hardware after initiation, it does give rather greater control in unusual applications. However, the hardware does provide completely transparent DMA transfers and dynamic memory refresh (itself performed by DMA). A DMA request will be honoured by the CPU as soon as it executes a microinstruction which specifies the default IDLE bus control function. Should the CPU then encounter a microinstruction specifying a non IDLE function after control of the bus has been given to a DMA device it will be suspended until the bus is released. Because of the hardware interlocks the microprogrammer is never aware of any device other than the CPU in control of the bus.

The system bus interfaces to the D bus in the CPU by two 32-bit registers known as the input and output bus registers. The output bus register is often referred to simply as 'the bus register'. Figure 7.1 is an illustration of this. The output bus register must be loaded explicitly by the microprogrammer with either an address or a word of data as appropriate for the bus function to be performed, whereas the input register is loaded automatically by the hardware during read functions. A one bit field in the control word controls the loading of the output bus register. Specifying the mnemonic LBR in a

microinstruction sets this bit and causes the output bus register to be loaded from the D bus. Thus, to move the contents of R3 into the bus register we could do

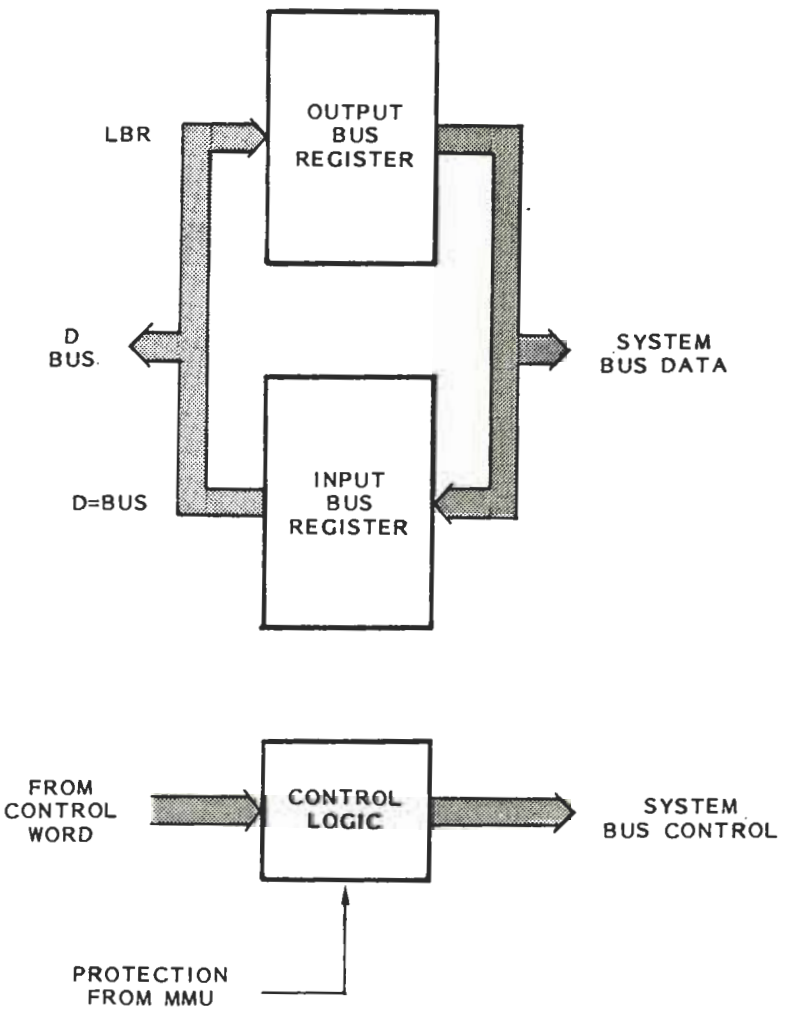
```
CONT  ZA A=R3  OR  LBR
```

At the end of the cycle the value will be loaded and will remain in the bus register until it is reloaded by another LBR. Thus it does not have to be transmitted to the memory system immediately. To access data which has been stored in the input bus register by a read operation we use the D=BUS D bus control function. We could transfer newly read data from the bus interface back into R3 with

```
CONT  DZ D=BUS  OR  RAMF B=R3
```

As with the output register, data will remain available for reading in the input register until another bus read operation causes it to be overwritten.

Figure 7.1 The system bus interface.



7.4 The bus control functions

The bus control functions are listed in Table 7.1. Rather than describing each in detail here, we will begin with some typical sequences to perform read and write operations and then give the rules for generating more sophisticated examples. Consider the following sequence.

```
CONT  ZA A=R0  OR  LBR  ADDR
CONT  LOCK
CONT  RD
CONT
CONT  DZ D=BUS  OR  RAMF B=R1
```

This code treats the contents of R0 as a physical address, reads the contents that memory location, and places the result in R1. On the first instruction the address is moved into the bus register and the ADDR (address) bus control function is given. As has been said, the LBR does not load the data into the output bus register until the end of the cycle. The ADDR is given on the same instruction since, because of the pipeline delay in the memory control path, it will not actually be executed until the following cycle when the address will be present in the bus register. The ADDR places the content of the output bus register on the system bus data lines and causes each memory module or I/O device to take and store the address on those lines. If the address falls within the address range of a particular module then that module is said to be selected. After the ADDR a selected module will become active and will remain active until the next ADDR or IDLE function. (IDLE is the default and occurs on any cycle which does not explicitly contain any other bus control function.) The LOCK on the

Table 7.1 Bus control function mnemonics.

Mnemonic	Description
IDLE	Default, no action
ADDR	Send physical address
RADDR	Send address, catch in input register
LOCK	Lock bus during operation
WR LWR	Write to selected address
RWR	Write to odd address
RD LRD	Read selected address
RRD	Read odd address

second cycle serves to lock the bus against access by a DMA device. It must be used here because the memory module must be given time to perform a read cycle before the data can be transferred back to the CPU. The third cycle specifies RD (read) which causes the data transfer to take place. Remember that the bus control functions are delayed by a cycle so the actual transfer occurs during the fourth cycle. On the fifth we can move the data into R1.

In practical cases memory operations are almost always overlapped with other functions. This is possible because once the address has been moved into the bus register only the bus control field in the microinstruction is needed to complete the operation.

It is most important never to omit the LOCK during a memory read since that would allow a DMA device to take the bus and issue a new address. This would select a different module and the remainder of the bus sequence would be operating on an undefined address with disastrous consequences.

A typical sequence to perform a write to memory is

```
CONT  ZA A=R0  OR  LBR  ADDR
CONT  ZA A=R1  OR  LBR  WR
CONT  LOCK
```

which writes the contents of register R1 to the address in R0. Here the WR (write) function can occur straight after the ADDR since we do not need to wait for the memory module to complete a read access when writing data. The LOCK is needed because the module must be active for at least two cycles or its operation is undefined. An acceptable alternative would be

```
CONT  ZA A=R0  OR  LBR  ADDR
CONT  LOCK
CONT  ZA A=R1  OR  LBR  WR
```

in which the write has been deferred. As we will see this flexibility is often useful when the data has to be computed after the address. It may not be immediately available.

Although a memory operation must not be shorter than two cycles, it can be longer. The upper limit on the cycle time is 8us which is 40 microinstructions even at the slowest clock speed (see chapter 10 for a discussion of clock speeds) and is unlikely to be exceeded in practice. It is not good practice to LOCK the bus for longer than is really needed since this prevents any DMA activity. Longer cycles often occur in read/modify/write operations. For example, suppose we wish to increment the contents of the memory location addressed by R0. We can say

```
CONT  ZA A=R0  OR  LBR  ADDR
CONT  LOCK
CONT  RD
CONT  LOCK
CONT  DZ D=BUS  ADD CIN RAMF B=R1  LOCK
CONT  ZA A=R1  OR  LBR  WR
```

which causes the same memory location to remain active whilst the

modification takes place and finally writes the result back. We have used R1 as a scratch register on the assumption that the address might be needed again later. As before omission of any of the LOCKs would be a serious error.

Although it might appear meaningful, the converse operation of performing a write and reading the data back in the same operation will not work. Undefined data will be returned by the RD in the following example.

```
CONT  ZA A=R0  OR  LBR  ADDR
CONT  ZA A=R1  OR  LBR  WR
CONT  RD /* Reads undefined data */
```

In all of the above examples we have been accessing a single memory location and the address involved can be either even or odd. We will now consider cases in which both words of a double word pair are read or written. For these examples to work it is vital that only even addresses are used. If an odd address were to be given then the same word would be accessed twice rather than the two halves of a pair. Two additional functions are provided for double word operations. These are RRD (right read) and RWR (right write) which act upon the right (odd) member of a double word pair. For clarity in these examples we also make use of LRD (left read) and LWR (left write) which are synonyms for the RD and WR functions that we have been using. These act upon the left (even) member of the pair.†

First we will read a double word into R1 and R2, again taking the address from R0.

```
CONT  ZA A=R0  OR  LBR  ADDR
CONT  LOCK
CONT  LRD    /* Equivalent to RD */
CONT  RRD
CONT  DZ D=BUS  OR  RAMF B=R1
CONT  DZ D=BUS  OR  RAMF B=R2
```

If this seems slightly odd remember that the action of the bus control function is delayed so that the first word is not available in the input bus register until the penultimate cycle. The first word must be read on that cycle or it will be overwritten with the data from the RRD. As before, additional LOCKs can be inserted to delay the data transfer if the data cannot be read immediately. Further, the order of the LRD and RRD can be interchanged, or the LRD can be omitted if only the odd word is wanted. We see in this example how a double word access takes only one cycle longer than a single one.

---

† LRD and LWR act upon the location specified by the address transmitted by the ADDR. RRD and RWR cause the LSB of the transmitted address to be set. Hence, if the address is even, the L functions access the even word and the R functions the odd one. If it is odd, both will access the odd word.

The corresponding double word write might look like

```
CONT   ZA A=R0  OR  LBR  ADDR
CONT   ZA A=R1  OR  LBR  LWR /* Equivalent to WR */
CONT   ZA A=R1  OR  LBR  RWR
```

Here we have been able to drop the trailing LOCK since the active part of the memory cycle is now automatically at least two cycles long. As in the read case we can insert LOCKs if required or reorder the LWR and RWR.

A more interesting example is a memory sequence which implements the CONS operation of LISP. We assume that the CONS cells are implemented as double word pairs and that a register called 'freelist' points to a list of free cells chained on the odd word. If the items to be stored in the newly claimed cell are currently in registers R0 and R1 then the following is all that is needed.

```
CONT   ZA A=freelist  OR  LBR  ADDR
CONT   ZA A=R0  OR  LBR  LWR
CONT   RRD
CONT   ZA A=R1  OR  LBR  RWR
CJP    Z,gc    DZ D=BUS  OR  RAMF B=freelist
```

We begin by addressing the first item on the freelist and write the contents of R0 to the even word (the CAR of the cell). Before writing to the odd word (the CDR of the cell) we must first read it to obtain the new freelist pointer. This is the purpose of the RRD on the third instruction although as usual we cannot actually use the data until two cycles later. On the fourth cycle R1 is written to the odd word. The last cycle inspects the value of the new freelist pointer for zero which would indicate the need for some form of garbage collection. Although a write is followed by a read here they act on different words; thus the previous restriction does not apply.

It is permissible to start a new memory operation immediately after a previous one has completed. The only restriction is that the bus should not be locked for more than the recommended 8us maximum. Thus, we could move a block of 16 words from the address in R0 to the address in R1, assuming both of these to be even, with

```
PUSH   7
CONT
       CONT   ZA A=R0  OR  LBR  ADDR
       CONT   ZA A=R1  OR  LBR  LOCK
       CONT   DA D=BR,2 A=R0  ADD  RAMF B=R0  LRD
       CONT   DA D=BR,2 A=R1  ADD  RAMF B=R1  RRD
       CONT   DZ D=BUS  OR  RAMF B=R2  ADDR
       RFCT   ZA A=R2  OR  LBR  LWR
       CONT   D=BUS  LBR  RWR
```

The loop body (the indented region) will be repeated eight times and each iteration moves two words. Several interesting things should be noted from this example. On the first cycle the address in R0 is moved to the output bus register ready for the ADDR. On the second, the other address is moved to the output register. It is done here rather than later because the D bus will be in use then reading the first word

of data. The addresses are incremented by 2 whilst waiting for the memory read to occur. When the first word of data is available it is moved into the temporary register R2. It cannot be moved directly into the output bus register as that would overwrite the second address which is yet to be sent. On the other hand, we cannot delay reading the data since the second data word would overwrite that if we did. On the last cycle, the second data word is moved directly between the input and output bus registers.

For the entire duration of this loop the bus is locked by the CPU. If a larger block is to be moved then the loop will have to include an IDLE to allow DMA devices (in particular the memory refresh controller) to use the bus. The next example does this and uses a register to control the number of loop iterations.

```
@1:     CONT  ZA A=R0  OR  LBR  ADDR
        CONT  ZA A=R1  OR  LBR  LOCK
        CONT  DA D=BR,2 A=R0  ADD  RAMF B=R0  LRD
        CONT  DA D=BR,2 A=R1  ADD  RAMF B=R1  RRD
        CONT  DZ D=BUS  OR  RAMF B=R2  ADDR
        CONT  ZA A=R2  OR  LBR  WR
        CJP   NZ, @1  ZB SUBR  RAMF B=count  D=BUS LBR RWR
        CONT  /* IDLE on this cycle */
```

The only bus control function which we have so far not discussed is RADDR. This is equivalent to ADDR except that the address is loaded into the input bus register while it is being sent down the system bus. It is provided primarily for diagnostic use to allow the bus to be tested, but it can be used to turn the input and output bus registers into a temporary storage register. The instruction

```
CONT  ....   LBR  RADDR
```

loads the contents of the D bus (determined by the unspecified part of the instruction) into the output bus register. On the next cycle it will be sent as an address and loaded into the input bus register, after which the input register can be read to retrieve the value. The fact that the data is sent as an address is harmless; no device will become active, even if accidentally selected, provided that the next bus control function is IDLE or ADDR (or another RADDR). The use of the bus registers in this way is discouraged because the CPU may be stopped temporarily if the bus is in use when the RADDR is executed.

Finally we will consider the restrictions which apply when constructing arbitrary bus control sequences.

A sequence must start with either ADDR or RADDR.

An ADDR or RADDR begins a new sequence.

An RD or RRD must not occur on the cycle immediately following ADDR or RADDR.

A sequence must not have a single active cycle (RD, RRD, WR, RWR, or LOCK) after the ADDR or RADDR. (An ADDR or RADDR in isolation is allowed.)

A read operation must not be done on a word which has previously been written by the same sequence.

Contiguous sequences with no intervening IDLEs should not extend for more than approximately 8us (40-50 microinstructions).

CHAPTER 8

Virtual Memory Management

Some form of memory management is essential in a system which supports multiprocessing. The primary reason is to provide protection of one process from another. Thus it should be impossible for a program in one process to damage another in a different process even if the former contains serious programming errors. The protection mechanisms must be implemented at a level below that of the programs to be protected in order that they be invisible and secure. Protection can be implemented with microcode, with special hardware, or with a combination of the two. Many factors are involved in arriving at the optimum solution. For example, a system dedicated to a language such as LISP may need no special hardware since the language provides its own memory management and protection mechanisms. In contrast, a system such as UNIX† which makes heavy use of the C language would be unusable on a system without some form of memory management.

Another major advantage of a memory management scheme is that it can provide a uniform environment in which to run multiple processes. That is, programs can be compiled and linked on the assumption that they will always operate at the same 'logical' memory addresses even though they occupy different physical locations when they run. All memory addresses generated by a running program are translated into the appropriate physical addresses before being used, without the program being aware that this is taking place.

The ORION memory management unit (MMU) is a tradeoff between performance, complexity, and cost. It has been designed on the assumption that most users will make use of the UNIX system and languages such as C, at least for development work. However, it is much simpler than in many machines and if appropriate it can be disabled and totally ignored. We assume here that the memory management hardware is being used, so that the primary purpose of this chapter is to show how new microcode can be written (possibly in the form of a new instruction set) to work within the framework of the standard system. We will then go on to consider memory management as seen by the operating system.

8.1 Virtual addresses

When memory management is in use all addresses generated by high level programs are virtual (logical) addresses. These must be translated into physical addresses before being sent over the system bus to the memory system. In the previous chapter we were concerned solely with the operation of the physical memory system and all the addresses referred to there were physical addresses.

---

† UNIX is a trademark of Bell Laboratories.

The primary component in the memory management hardware is a fast memory internal to the CPU which is used as a look-up table to perform the translation. This memory is known as the translation buffer; virtual addresses are presented to the translation buffer in a register known as the virtual address register (VAR). The output of the translation buffer is the physical address corresponding to the logical address in the VAR, or an indication that the address in the VAR is in some way invalid. Figure 8.1 shows a simplified diagram of the MMU.

The loading of the VAR is controlled by a single bit in the control word. Specifying the mnemonic LVAR in a microinstruction sets this bit and causes the VAR to be loaded from the AY bus. We could move a logical address from, say, R0 into the VAR with

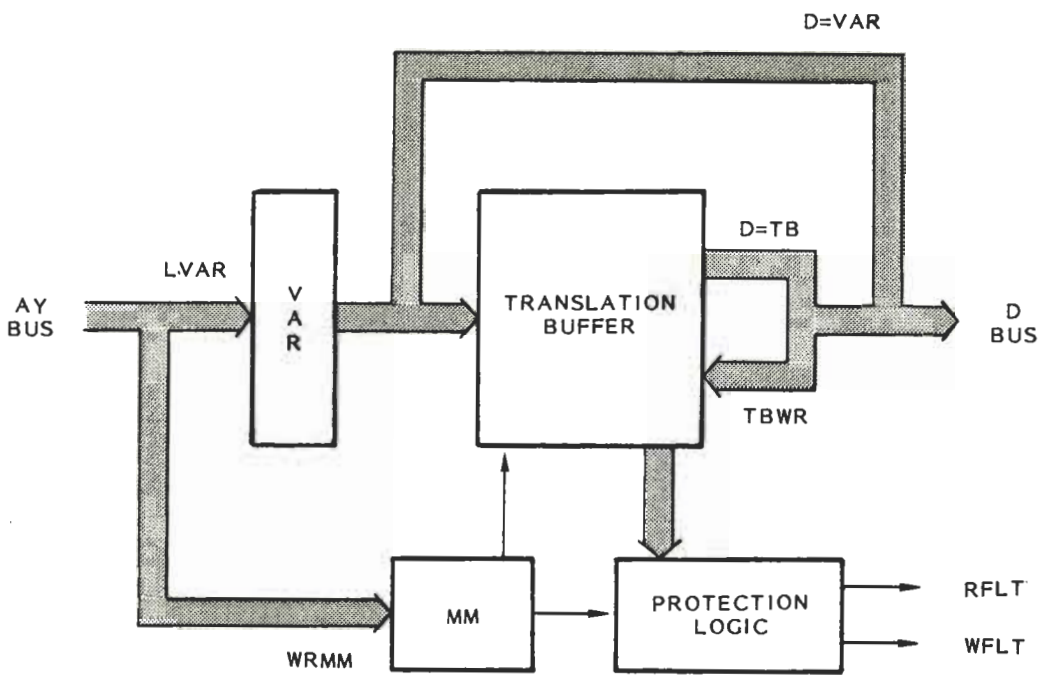
```
CONT   ZA A=R0  OR  LVAR
```

The translation buffer requires one cycle in which to perform the address translation. Thus, at the end of the following cycle the physical address will be available for loading into the output bus register. The output of the translation buffer is made available on the D bus using the D=TB D bus function. We can extend the previous example to complete the translation as follows

```
CONT   ZA A=R0  OR  LVAR
CONT   D=TB  LBR  ADDR
```

in which the physical address is loaded into the output bus register at the end of the second instruction. Although loading the address into

Figure 8.1 The memory management unit.



the bus register uses the D bus it does not use the ALU so that in real cases we would expect to find considerable overlap of the address translation with other operations in the CPU. From the second instruction onwards the memory reference can proceed exactly as in the previous chapter; the only difference is that the physical address has been provided by the translation buffer rather than direct from a high level program.

The virtual address register will retain its content until reloaded by another LVAR. Likewise, the output of the translation buffer will remain fixed if the content of the VAR is not changed. Hence it is not necessary to load the physical address into the bus register on the cycle immediately following the LVAR if this is not convenient. However, in operations which reference memory it is usually the memory access which is the limiting factor in deciding the minimum number of cycles required, so it is generally desirable not to delay.

8.2 Translation faults

We mentioned earlier that the translation buffer provides either a physical address corresponding to the logical address in the VAR, or else an indication that the address in the VAR is in some way invalid. So far we have been assuming that the logical address has been valid and that the translation buffer has indeed provided a physical address. In general, it is the responsibility of the microcode to check for an invalid address whenever a memory reference takes place and the MMU hardware provides the necessary support to make this straightforward. Each entry in the translation buffer contains, along with the physical address, an address tag and six of bits of protection information.† The state of these bits is set by operating system software and indicates such things as an invalid address (no physical memory allocated to this address), or read-only (writing to this address not permitted). This information is distilled by the hardware into just two condition codes, only one of which needs to be tested in any particular case, to indicate that some abnormal condition exists. When this happens, the microcode must take special action which may result in a trap back to the operating system before execution can continue.

In the simplest system a fault condition would indicate a fatal error caused by an incorrect program. In this case there would be no corrective action to perform and a trap would be the only option. In a more sophisticated system, supporting demand loading or demand paging, it would be necessary first to undo any actions which have been performed by the high level instruction prior to the fault being detected. This would allow the instruction to be retried after the fault has been corrected by the operating system.

The condition codes are RFLT and WFLT (together with the complementary conditions NRFLT and NWFLT). RFLT indicates that the current logical address (the address in VAR) points to an unreadable location; this condition should be tested whenever a memory access which performs only read operations is attempted. WFLT indicates that

---

† One of these bits is actually stored elsewhere.

the current logical address points to an unwritable location and should be tested when a memory operation which performs a write access is attempted. Thus WFLT is the correct condition to check in a read/modify/write operation.

Since the RFLT and WFLT conditions can only be generated after the translation has been performed, they cannot be tested until two cycles after the LVAR. To illustrate, we could complete the earlier example as follows

```
CONT   ZA A=R0  OR  LVAR
CONT   D=TB  LBR  ADDR
CJP    RFLT, trap  LOCK
CONT   RD
```

where the code at the label trap is assumed to take all necessary action in the event of a fault. The increment example of the previous chapter would become

```
CONT   ZA A=R0  OR  LVAR
CONT   D=TB  LBR  ADDR
CJP    WFLT, trap  LOCK
CONT   RD
CONT   LOCK
CONT   DZ D=BUS  ADD CIN  RAMF B=R1  LOCK
CONT   ZA A=R1  OR  LBR  WR
```

8.3 The logical address space

So far we have covered how to perform address translation without giving any information as to the detailed form of a virtual address or the precise contents of translation buffer entries. This information is only of interest when writing microcode to support the operating system functions of setting up translation buffer entries and handling faults. We now go on to consider the detailed operation of the MMU. The remainder of this chapter could well be skipped at a first reading.†

It is often desirable to be able to allocate memory resources to programs dynamically as they are run. This avoids wastefully allocating memory which will never be used or having a program fail because insufficient memory was initially allocated. In general it is helpful if the address space of a program can be grown independently in more than one place. For example, a C program in the standard system has two stacks and a heap, any of which may need to be extended. To facilitate this, the logical address space is divided into four identical regions distinguished by the two most significant bits in a virtual address (bits 30 and 31). Corresponding to these four regions there are four independent look-up tables in the translation buffer and the two most significant bits of the VAR determine which table will be used to perform the translation.

---

† It was skipped at first writing!

Each region has a single protection bit which allows the entire region to be enabled or disabled simply by changing that bit. This is exploited in the standard system to protect the kernel of the operating system. The kernel resides in region 3, which is disabled when user programs are running, but enabled when a system call is made.

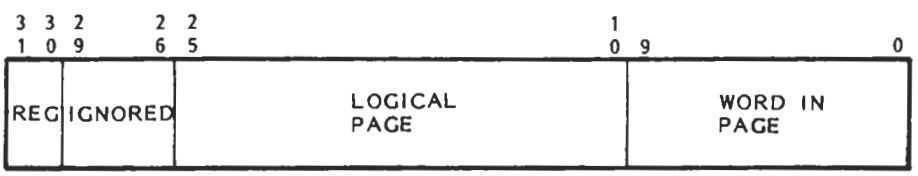
Within a region the address space is divided into equal sized 'pages' each of 4 Kbytes. A page is the smallest unit of memory which can be allocated or protected. Figure 8.2 shows the format of the full virtual address

At the least significant end 10 bits are used to specify a word within a page. These bits are unaffected by the address translation process; the D=TB D bus function takes these bits direct from the VAR. The next 16 are the logical page number and are used to index into the appropriate translation buffer look-up table. Each of these look-up tables contains only 512 entries and so only the 9 low order bits of these 16 are actually used as the index. The remaining 7 bits are compared against a tag stored in the selected entry and a translation invalid fault (see below) will be generated if any of them differ. This allows one entry to serve as a cache for more than one logical page.

Bits 26 to 29 are totally ignored by the hardware when the VAR is loaded. This is quite deliberate, and allows these bits to be used for other purposes. For example, the standard system must support byte addressing for the C language. This is accomplished by using bits 26 and 27 to indicate the byte within the word to which the address refers. Because the hardware is intrinsically word oriented this is less inconvenient than might be expected; the byte address can be loaded directly into the VAR and used as a word address. While the memory cycle takes place the microcode tests the two byte bits (using the OB and OS condition codes) to determine which byte to use when the word is available. It is anticipated that these bits will find other uses in different applications, such as garbage collector mark bits or pointer tags.

As has been said, the most significant two bits identify the region. When the VAR is loaded using LVAR, in addition to being stored in the VAR to select one of the four look-up tables, these bits are also used to index into a four word by four-bit table known as the

Figure 8.2 Virtual address format.



memory mode table (MM). The four bit word read from this table contains the region fault bit. This bit is latched along with the virtual address and is used by the protection hardware to enable or disable the entire region. The remaining three bits of the MM word are ignored by the hardware but we mention them here because they can be read and written and could be used for some other purpose by the microcode.

8.4 Translation buffer entries

Each entry in the translation buffer provides a 16 bit physical page number which is placed on D bus lines 10-25 when the D=TB function is specified. The entry is concatenated with the 10 word within page bits direct from the VAR to form the full 26-bit physical address. At the same time the six protection bits for this entry appear on D bus lines 26-31. These bits are ignored in physical addresses but are made available on the D bus so that they can be tested if a fault is detected, and so that the translation buffer entries can be loaded and unloaded as data transferred to and from the D bus. Figure 8.3 shows the format of a translation buffer entry. The intended uses of the protection bits are as follows

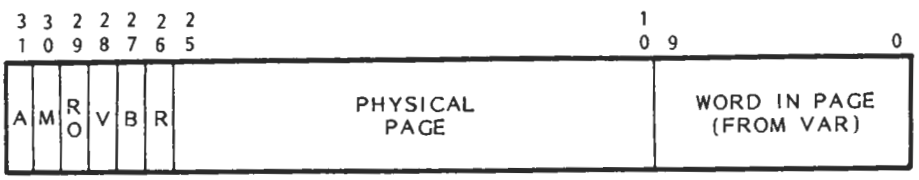
A (accessed) is set by microcode to indicate that a page has been accessed and that the corresponding translation buffer entry has been cached from main memory. It is cleared by software. This is intended to support a least recently used (LRU) paging algorithm. The hardware will make this bit appear reset if the address tag stored in the translation buffer entry does not match bits 19 to 25 in the VAR. This allows microcode to recache the correct entry.

M (modified) is set by microcode to indicate that one or more words in a page have been modified. It is cleared by software when the page has been written back to disk. This allows the paging algorithm to write out only those pages which have been modified.

RO (read only) is set by software to prevent write operations to a page. The executable text of a program will normally have the RO bit set to prevent and detect inadvertent modification.

V (valid) is set by software to indicate that a page is currently resident in physical memory. If it is not set then software intervention is needed to allocate physical memory and load the page.

Figure 8.3 Translation buffer entry.



B (bounds) is set to indicate that a page is beyond the end of the currently active portion of a region. On context switches all entries in the translation buffer must be invalidated by microcode by clearing the accessed bit for each entry. The bounds bit allows optimisation of this by indicating the first previously unused entry.

R (region fault) is set to indicate that the entire region is disabled. This bit is not actually stored in the translation buffer but is placed on the D bus when a translation buffer entry is read in order to simplify the tests which the microcode must perform. This bit is actually stored in the MM table and is ignored when a translation buffer entry is written (see WRMM, below).

The RFLT condition code will be true when any of the following conditions occur. They are listed in decreasing order of severity.

A translation buffer parity error has been detected.
The region fault bit is set for the selected region.
The bounds fault bit is set.
The valid bit is not set.
The accessed bit is not set or there is a tag mismatch.

The WFLT condition code will be true when any of the following conditions occur

A translation buffer parity error has been detected.
The region fault bit is set for the selected region.
The bounds fault bit is set.
The valid bit is not set.
The read only bit is set.
The modified bit is not set.
The accessed bit is not set or there is a tag mismatch.

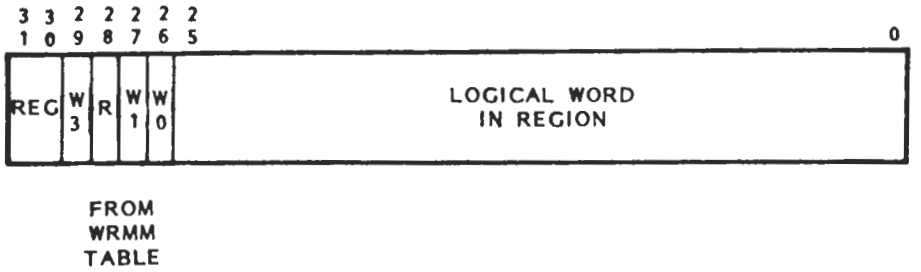
If a fault occurs, the hardware will convert a read or write bus control function into a LOCK to prevent the access from occurring. This is particularly important because of the instruction pipeline since otherwise the microcode would have to complete the test for a fault before proceeding with the memory operation. To illustrate, consider the following fragment

```
        CONT   ZA A=R0  OR  LVAR
        CONT   D=TB  LBR  ADDR
        CJP    WFLT, @1  ZA A=R1  OR  LBR  WR
        CONT   LOCK
          .
          .

@1:     /* Deal with fault */
```

If, for example, the valid bit is not set then the WR bus control function in the instruction containing the WFLT condition code will be converted into a LOCK and no memory location will be modified. Without the hardware intervention it would have been necessary to defer the WR by two cycles (by inserting LOCKs) to give the branch time to occur.

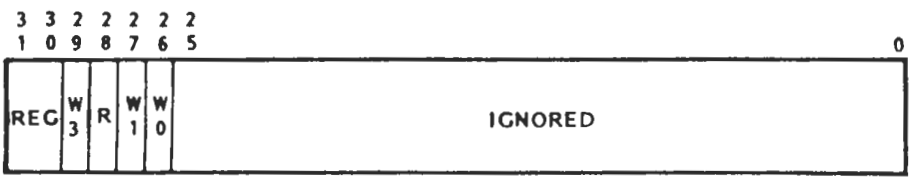
To assist the microcode in dealing with addressing faults the VAR can be read back onto the D bus using the D=VAR function. This allows fault handling code to inspect the failing virtual address. When the VAR is read in this way bits 26 to 29, which are ignored when the VAR is loaded, contain the four bit word from the MM table. This allows the state of the region fault bit to be tested. (Region fault is also read by D=TB.) Figure 8.4 illustrates this.

Figure 8.4 Reading the virtual address register.



The MM table is written using the WRMM† special function. This function uses the two most significant bits of AY to select the region and hence which of the four entries in the MM table to write. The contents of bits 26 to 29 of AY are then written to the selected table entry. Bit 28 is the region fault bit and setting it disables the region. Bits 29, 27 and 26 are stored but are otherwise ignored by the hardware. All four bits will appear on the corresponding four bits of the D bus when D=VAR is specified. The operation of the fault logic is undefined if LVAR and WRMM occur in the same instruction. This condition should be avoided.

Figure 8.5 Writing to the MM table.



Translation buffer entries are written using the TBWR special function. The entry to be updated is selected by first loading the corresponding logical address into the VAR and on a following cycle issuing a TBWR with the data to be written on the D bus. The format of the data should be as shown in Figure 8.3, except that bit 26 and the word in page bits are ignored. The tag field of the translation buffer entry is not directly accessible to the microprogrammer. (Hence it has not been shown in Figure 8.3.) Bits 19 to 25 of the VAR will be

† WRMM is pronounced worm.

stored in the tag field by the hardware during a TBWR for later comparison by the fault logic. A subsequent tag mismatch can be treated exactly as if the accessed bit had not been set without any need to know the actual value of the stored tag.

So, for example, to update the entry corresponding to the logical address in R0 with the value in R1

```
CONT   ZA A=R0  OR  LVAR
CONT   ZA A=R1  OR  TBWR
```

The operation of the fault logic is undefined on a cycle in which a TBWR is performed. The RFLT and WFLT conditions should not be tested on such a cycle.

8.5 Use of the memory management unit

The simplest way to employ the translation buffer is to have the operating system load it with an appropriate memory map before dispatching a process. All entries corresponding to accessible logical addresses should have their modified, accessed, valid, and optionally, their read only bits set. Inaccessible logical addresses should have their bounds bit set. Any kind of fault which is subsequently detected indicates an access violation and control should be passed back to the operating system for high level code to intervene. Although this is conceptually the simplest use of the translation buffer, it suffers from the disadvantage that the process dispatch operation will be rather slow because of the need to completely reload it and the size of processes will be limited by the number of available translation buffer entries to 2 Mbytes per region. This may be acceptable in a simple single process system which does not need to support demand paging.

A more sophisticated application is to be found in the standard system where the translation buffer is treated purely as a cache onto memory based translation tables. The accessed bit is used to indicate an invalid translation buffer entry when it is not set. When such an entry is referenced a fault is indicated and the entry is loaded in from the translation table in memory. This is handled entirely in microcode and control is only passed back to high level code for more serious faults. At process dispatch time the microcode now only has to clear the accessed bit in translation buffer entries which were previously active. This can be done much more quickly than reloading completely from memory. If a process does not run for long before the next dispatch, or if it makes only localised references to memory, few translation buffer entries will be read in.

When an entry is loaded in from memory the accessed bit is set in both the translation buffer entry and in the copy in memory. Similarly, if a WFLT occurs because the modified bit is not set the microcode sets the modified bit in both the translation buffer entry and in the copy in memory.

After the accessed bit or the modified bit of a translation buffer entry has been set, subsequent references to that page will not generate a fault. The accessed and modified bits held in the memory based table are used by the operating system to improve the virtual memory

paging performance. If set, the modified bit indicates that the contents of a page have been changed; the page must be written to the paging device before the physical memory page can be reallocated. The operating system periodically clears the accessed bits. When a new page is required, one whose accessed bit has not been set is chosen. This will be a page which has not been referenced recently.

CHAPTER 9

The Diagnostic Microprocessor

The CPU contains an embedded microprocessor (DP) whose main role is to load the control store and map tables at the time of bootstrap. Since the control store is implemented entirely in RAM the main CPU cannot run until an initial microprogram has been loaded. When the system is running the DP provides the facility to load or unload parts of the control store and map tables dynamically. This is particularly useful during microcode development as new code can be loaded into a running system.

The second major role of the DP is to control the execution of diagnostics when a system failure occurs. We will not consider this use in detail here; however, some of the facilities provided in the DP firmware for diagnostic use can be of value during microcode development and debugging and these will be covered in this chapter.

9.1 Bootstrapping

After power on the DP performs a number of static confidence checks on the CPU. It then loads a microprogram into the control store and map tables to perform a more extensive check of the CPU. If any of these tests fail the DP flashes the front panel LED and aborts the bootstrap procedure. If all is well it proceeds as if there had been a front panel reset. A different microprogram is loaded which initialises the memory management hardware and investigates the physical address space. The microcode looks for an I/O controller (IOC) waiting to load the full microcode and operating system from disk or tape. It then reads a file of combined microcode and DP code from the IOC and hands it back to the DP to be loaded into the control store or its local RAM. The DP keeps a copy of any microcode which would overlay the bootstrap and loads this in last. When the IOC has loaded the operating system by DMA it signals to the DP.

In the standard system the downloaded DP code starts the microcode running and then enters a support loop processing commands from the CPU. These allow the control store and map tables to be loaded or unloaded dynamically, and allow diagnostic error messages to be sent from the CPU to DP serial port 0

9.2 DP/CPU communication

A bytewide communication path exists between the CPU and the DP. On the DP side it is assumed that control programs are written in BCPL and library procedures are provided to transfer data through the port. On the CPU side the data port actually appears to be the L section of the instruction register IR. Two special functions ININT and OUTINT are provided to accomplish the transfer, together with two status bits to indicate the DP's readiness in the two directions. These bits are

read as bits 14 (OUTRDY) and 15 (INRDY) on the D bus when the D=CAIR function is specified.

INRDY indicates that a byte is waiting to be read by the CPU. The following sequence should be used

```
byte.in:

/* Input byte returned in R0 */
/* Test ready bit (bit 15 in CAIR) */

        CJP     NS, $  DZ D=CAIR SHL2  OR
        CONT

/* Allow time for port to turn on */

        CONT   ININT
        CRTN   ININT
        CONT   DZ D=CAIR  ZZZD  OR  RAMF B=R0  ININT
/*--------*/
```

The ININT generates an interrupt to the DP to indicate that the byte has been accepted. Note that it is repeated to match the leisurely pace of the DP. After the ININT the byte is read from the IR with the D=CAIR function. Depending on the application it may be necessary to save the IR and restore it.

The corresponding output operation is accomplished by loading the byte to be output into the IR when the OUTRDY bit is set. Then the OUTINT special function is used to interrupt the DP. The following sequence should be used

```
byte.out:

/* Least significant byte of R0 output */

        CONT   DZ D=BR,40H SHL1  OR  RAMA A=R0 B=R0  LDIR

/* Test ready bit (bit 14 in CAIR) */

        CJP     NZ, $  DA D=CAIR A=R0  AND
        CONT

/* Allow time for strobe */

        CRTN   OUTINT
        CONT   OUTINT
/*--------*/
```

Again the multiple OUTINTs are to allow time for the DP to respond.

Because only polled I/O is possible this path is not suitable for high speed data transfer. However it is quite adequate for the dynamic control store loading most useful during microcode development.

9.3 Diagnostic terminal functions

By attaching a terminal to DP serial port 0 it is possible to interact directly with the DP. This serial port operates by default at 9600 baud with XON/XOFF protocol, but with hardware handshaking also available. Serial port 1, which by default operates at 19200 baud, can be used for the downloading of diagnostic programs. For this purpose it would normally be linked to one of the serial lines on an IOC. From the diagnostic terminal (DT) microcode can be single stepped, examined, or its execution traced.

To gain the attention of the DP at any time type CTRL C, when it will respond with the prompt

DT>

Note, however, that this should not be done when the system is running as the CPU will be halted immediately. Commands can now be typed consisting of a single letter optionally followed by one or more numeric or character string arguments. Numeric arguments are taken to be in base 10 unless the first digit is 0 in which case they are taken to be in base 16. Commands are acted upon after the RETURN key has been typed. The '?' command will cause the DP to list all the other available commands.

There is a line editor which allows corrections to be made before RETURN is typed. DELETE and CTRL H delete a single character. CTRL U deletes the entire line, while CTRL R redraws the line.

The commands available are as follows

A Enter auto-boot mode. This is the usual state after the power on confidence check or a front panel reset. In this state anything except CTRL C typed on the diagnostic terminal will be ignored.

B Begin execution. The CPU clock is started. A previous G command should have set the initial microcode addresses.

C Run confidence check. The power on CPU confidence check microcode will be loaded and run. Any failure will be reported.

D <filename>
Download a file. This command is intended for loading diagnostic programs. The format of the string <filename> is dependent on the host system connected to serial port 1.

F Select fast clock. This is the default state after power on or reset. The CPU clock runs at the full rate. This is used at all times except during hardware fault diagnosis.

G <n> <m>
Set the starting addresses prior to commencing execution. Two addresses are required because of the microinstruction pipeline. The instruction at address <n> will be executed first and that at address <m> second. A symbolic disassembly of the first microinstruction will be displayed. <n> and <m> default to 1 and 0 respectively if omitted. These are the conventional starting addresses used by the standard system.

H Halt execution. The CPU clock is stopped.

L <n> <m>
Disassemble the control store between addresses <n> and <m> inclusive. If <m> is omitted it defaults to <n> and a single line is listed. Output can be interrupted at any time by typing CTRL C.

M <n> <m>
Display map table entries between <n> and <m> inclusive. If <m> is omitted it defaults to <n> and a single entry is displayed. Output can be interrupted at any time by typing CTRL C.

O <baud>
Set the baud rate of serial port 1. If <baud> is omitted the current baud rate will be displayed. This allows diagnostics to be downloaded from something other than an IOC.

P <n> <m>
Access DP I/O ports. The value <m> is written to the DP I/O port at address <n>. If <m> is omitted the port at address <n> will be read and its content displayed. Intended for diagnostic use.

R Run a downloaded test program. This is intended for diagnostic use and causes the DP to pass control to the downloaded program.

S Select slow clock. The CPU clock will run at approximately half its normal rate when started. This is provided for use during hardware fault diagnosis.

T <addr> <n>
Microcode is single stepped until the microinstruction at address <addr> is encountered. This is repeated <n> times. Execution stops with a symbolic display of the instruction at address <addr>. <n> defaults to 1.

U <n> <m>
Access DP I/O memory. The value <m> is written to DP memory at address <n>. If <m> is omitted the memory at address <n> will be read and its content displayed. Intended for diagnostic use.

Z <n>
Microcode is single stepped <n> times. At each step the current instruction is displayed. <n> defaults to 1. Note that it is not possible to single step though microcode which makes memory references. This is because the system bus cannot be locked for extended periods. If an attempt is made to do this several steps will be taken until an IDLE bus control function is encountered. As a result, the instruction symbolically displayed will be incorrect.

CHAPTER 10

Advanced Techniques

In this chapter we will take a look at some rather more advanced examples of microcode, drawn from the standard instruction set. The examples are presented in the form of source code listings and are to be found in Appendix E which should be read in conjunction with this chapter. Although Appendix E contains only a few small excerpts from the standard instruction set they have been chosen to illustrate both interesting techniques, and the requirements which custom microcode must meet if it is to be added to the standard system. Appendix F illustrates how a new piece of microcode can be assembled, linked and added to the system for testing.

10.1 Included files

Commonly used definitions are collected together into header files which can be included in several different source files. This allows a consistent set of definitions to be used by all the modules in the system. A file is included with the microassembler *I directive. For example

```
*I register.m
```

Many such directives will be found in the listings.

10.2 Register definitions

The most common header file, which is included by all the modules of the standard system, is register.m. This file defines mnemonic names for several of the ALU registers and we refer to the listing in the following discussion.

Registers R0, R1, R2, R3, and R4 are used as scratch registers. They never contain meaningful values between one high level instruction and the next. The 0 register is also used as a scratch register but it is not mentioned in this file because it is accessed using entirely separate functions in the control word as compared to the ordinary registers. ir0 and ir1 are used as the eight byte instruction prefetch cache. Code is fetched from memory two words at a time in order to exploit the interleaving of the main memory. ir0 contains the low word of a pair, which will be executed first, and ir1 the high word. pr is a pointer to a buffer to be used when dynamic instruction profiling is enabled. Keeping the pointer in a fast register greatly speeds up profiling. We will consider the profiling code later in this chapter. psw contains several independent pieces of information, in particular the current instruction set number in bits 8-11 and the interrupt enable bit in bit 0. cb is a pointer to the base of the stack region which is currently held in the cache memory. cb is used by the procedure call and return instructions to decided when to swap data between the cache and main memory, and by certain data referencing instructions which must decide whether to look in the cache or main memory. The

remaining registers are the program counter, stack pointers (scalar and vector) and frame pointers (scalar and vector) of the abstract C machine.

New microcode to be added to the standard instruction set must obey the same conventions concerning the use of the registers. If a large number of temporary registers is required, as in the floating point microcode, then some registers can be swapped out into an area of cache memory. If an entirely new instruction set is being constructed then there will be no constraints in the use of the registers. However, in such a case, early consideration should be given to the operating system call interface if the new instruction set is to run alongside the standard system.

10.3 Initialization

Our next example file, rinit.m, is the module which performs initialization of the hardware after power on. This microcode is actually stored in the ROM of the DP which loads it into the control store after running the confidence check. It illustrates the use of microcode loops and subroutines as well as many of the more obscure control word functions.

A header file is included; vahdr.m contains useful constants relating to virtual memory management.

Since this is the very first microcode to be executed after the system is started none of the special registers are in use. We define local names of greater mnemonic significance to be used during the initialization. Several other constants are defined here.

The first two actual microinstructions are the startup code. The DP begins execution at addresses 1 and 0. The JZ at address 1 is the very first instruction to be executed and it initializes the sequencer. The instruction at address zero is executed twice. This is because the DP explicitly sets zero as the second address and then the JZ transfers control there. Careful consideration of these two instructions should reveal that although the first is executed twice the branch only occurs once.

After power on all the memories in the machine contain random data which must be cleared in order to remove potential parity errors. We begin by clearing the cache. Nested loops are used, the outer one stepping over the H section of CA and the inner one over the L section. In fact, there are two inner loops, one to deal with the normal half of each bank and one to deal with the alternate half. Note that we write to all sixteen cache banks even though current implementations have only two. This makes the code independent of the actual size of the cache.

The alternate half of cache bank zero is used to hold a set of privileged registers used by the operating system to control memory management. At initialization we build a map in these registers of what is found in the physical address space. There are 512 registers in the alternate cache bank and 512 logical slots in the address space. Before probing the address space the MM table and the translation buffer are

initialized. We must do this in order to prevent the fault logic from inhibiting memory operations. Each entry in the MM table is set to have no region fault. All entries in the translation buffer are first written to set the bounds bit and remove any parity errors. Next the entry corresponding to logical address zero is written with a value containing no fault. The address of this entry is left in the virtual address register so that no faults will be indicated when physical addressing is used.

For each slot in the address space we write zero to the lowest location and read it back. Memory will read back as zero and an empty slot as -1. I/O devices have an identification register at the lowest address. This is an eight bit register (bits 0-7) which identifies the type of the device. On each iteration of the loop we use CLRPERR to clear any memory parity errors caused by accessing an empty slot. If memory is found, it is cleared by writing zero to all addresses in the slot. Then, entries in the translation buffer are initialized to point to the newly found memory. We map the same memory into all four regions. A maximum of 2 Mbytes is mapped in this way, any more simply being noted in the privileged registers. By setting up a mapping in this way we can begin execution of the operating system directly in the C language and perform the remaining initialization there. Further, the distribution of physical memory in the address space is unimportant.

After the memories have been initialized, the main microcode and the operating system can be loaded into memory. This is done by an IOC in the system using DMA to load the memory. The IOC also probes the address space to find the lowest physical memory and loads the system there. This will have been mapped into the beginning of logical region 3. First the microcode is read from the IOC and passed into the DP for loading into the control store. We make use of routines similar to the byte.in and byte.out procedures of Chapter 9. When the DP reads the end of the microcode file it will stop the CPU, overlay the rinit microcode, and start execution of the main system.

10.4 Instruction fetching

The standard system uses two of the Am2901C registers as an instruction prefetch buffer. Two words are fetched in one operation in order to exploit the interleaving of the main memory. As our next example we take a look at the microcode which is responsible for refilling the prefetch buffer when it has been emptied.

The two registers comprising the buffer are known as ir0 and ir1. Of these, ir0 holds the low order word and ir1 the high order one. Code is executed starting with the low order byte of ir0 and as each byte is consumed ir0 is shifted right bringing in a zero at the high order end. When the first of these zeros reaches the low byte and is loaded into IR, control is transferred via map table entry 0. This entry point is called fetch and is to be found towards the end of the first page of the listing of fetch.m Two entry points are defined together here

```
ENTRY umode + fetch
ENTRY kmode + fetch
```

which puts this code into two different instruction sets. umode is the instruction set in which user mode C language programs execute, whereas kmode is the instruction set of the operating system kernel. These two instruction sets are almost identical but certain special instructions are available in kmode which cause traps in umode. These are known as privileged instructions and allow the operating system to control memory management securely.

The Am2901C register referred to as pc is the machine program counter. It points to the word of code after the one currently being executed. If it is odd then ir1 contains a prefetched word of code which simply needs to be transferred into ir0. The least significant byte can then be moved into IR in the usual way. If it was even then the CJV fails and code has to be fetched from memory. The program counter is moved into the VAR and is incremented by one before the CJV has occurred. This allows the memory access to be started early in case it is needed. If code is being fetched from memory, a test is made for any parity errors having been flagged. If so a branch is made to except where the true cause is determined and a trap is taken. Next a test is made for an RFLT condition. Stop for a moment to consider the control flow in the two possible cases. Finally, the code from memory is loaded into ir0 and ir1 and the next instruction is decoded.

After any form of control transfer, such as a branch or procedure call, the value of pc will have been changed and any code in ir0 or ir1 must be discarded. Control transfer instructions use the FETCH function to load 1 into IR. Entry 1 in the map table is set by the refill entry which is at the beginning of the fetch.m file. Here code will always be fetched from memory even if the pc is odd. First however, a number of checks are made to test for parity errors and pending interrupts. If these fail (the usual case) control is passed to the same code we have already looked at. This will happen if either there is no interrupt pending (NINT condition code) or the LSB of register psw is cleared. The LSB of register psw (processor status word) is the interrupt enable bit. The interrupt is also ignored if a parity error has been flagged. Interrupts are accepted by performing a trap and passing an appropriate trapcode. Note the way that refill is moved into ir0 and the pc is decremented so that the fetch will be restarted after a return from the interrupt.

10.5 Stack manipulation

The file lp.m contains instructions which are typical of those which manipulate the scalar stack. The top of the scalar stack is where expression evaluation takes place and where storage is allocated for local scalar variables. The lp group of instructions access local variables in the current stack frame by addressing relative to the frame pointer (fp) and load their values onto the top of the stack (identified by the stack pointer, sp) for use in expressions.

These instructions occur very frequently, particularly for small values of the offset from the frame pointer. In order to keep the code

compact and fast these most frequently occurring cases are treated specially by providing a set of one byte opcodes in which the value to be used for the offset is implied by the opcode and embedded in the microcode for each case. This is the group lp3 to lp10. All of these are identical except for the value used for the offset.

Considering lp3 as a typical case, we begin by reloading IR with the next byte from the code stream. This must be done before the CJV. Then, the offset is added to the frame pointer and the result moved to the cache address register with the LDCA. On the following instruction the stack pointer is incremented to point to the new top of stack after the value has been loaded and this value is moved into the cache address register. The final instruction both reads and writes the cache in the same operation. This is exploiting the one cycle delay from the cache read pipeline; although the address has been changed by the second LDCA the value read is that from the location addressed by the first LDCA.

Note here that we obey the rule that within the standard system CA is never changed on the last cycle of a microinstruction, and that CA and sp are always left pointing at the top of the stack. This allows the following opcode to read the top item on the stack in its very first microinstruction if required.

When the offset is outside the range of the special cases it is encoded as a second byte in the opcode. This is quite a common occurrence throughout the instruction set and our next example is intended to be typical. The compiler ensures that the two bytes of the opcode are in the same word. If necessary a pad byte of zero will be inserted to fill the previous word. This pad byte should not be thought of as a no-op because the normal instruction decoding mechanism uses the value zero to indicate the need to refill ir0. The pad byte simply makes this occur one byte earlier than usual.

lp_w performs the same function as the special cases above but takes the offset from the next byte in the code stream. Note the use of PLDIR to reload IR whilst extracting the byte. ir0 is then shifted right by two bytes to remove both the parameter byte and the next opcode.

The following group of instructions in this file perform load operations for operands of types other than a machine word. Such operands are always aligned to word boundary in the stack so the only difference is the treatment of high order bits. lp_c loads a character, masking the high order 24 bits to zero. lp_s loads a short (16-bit) two's complement integer and performs sign extension. Note the trick used here to move the meaningful part of the value into the high order 16 bits temporarily so that the sign bit can be tested. lp_h loads an unsigned short integer and is straightforward. lp_d loads a double word. An interesting sequence of CA functions is used and sp is incremented by two in this case. lp_f loads a single precision floating point datum and converts it to double precision on top of the stack. This is required by the rules of expression evaluation in the C language. An external routine ftod is called to do the conversion.

Finally in this file we mention llp which loads the address of a local variable rather than its value. This just involves adding the parameter byte to the frame pointer and loading the result onto the top of the stack.

10.6 Memory references

To provide an example of a memory reference made in the context of the standard system we have included the file ln.m. This file contains operators to load constants onto the top of the stack. Only the last instruction in this file, lnw, is of real interest here. As with lp, a number of special cases are recognised for ln; however, a byte parameter can only cope with values in the range -255 to +255 so a new form is required in which the parameter occupies a full word. The word addressed nature of the underlying hardware dictates that this value should be aligned on a word boundary in the code stream. At first sight this might suggest that a large amount of space would be wasted in the form of padding to achieve the desired alignment. In fact, the standard system arranges that the value will be placed in the next available word in the code stream without padding. Subsequent one or two byte opcodes, or the opcode bytes of further instructions with word parameters are placed in any available space before the word boundary. Since the pc is advanced by one word each time code is reloaded into ir0 it will already be pointing at the next word in the code stream. This is the word containing the parameter of the current instruction. We can consume this word and increment the pc to point beyond it without affecting the contents of ir0. It may be worth devoting a few moments thought to what actually happens here, particularly in the case where several instructions with word parameters occur close together, before proceeding.

The first thing to be decided in an opcode such as this is whether the word parameter has been prefetched into ir1 or whether it must be retrieved from memory. This is done by checking the LSB of the program counter. An odd pc implies that the word has been prefetched into ir1. In this case the value can be used directly and all that remains is to increment the pc so that ir1 is not subsequently reinterpreted as code.

If the pc is even a memory reference must be performed. As with code fetching we arrange to start this before having decided if it will be needed. Two words can be fetched almost for the price of one here since the pc was known to be even. The first word read is used as the parameter; the second is loaded into ir1 to be used later as code.

As with any memory reference we must check for addressing faults. A subroutine called rcache is called conditionally if an RFLT condition is detected. By far the most frequent problem is simply an uncached translation buffer entry which can be fixed on the fly. rcache is careful to preserve the state of the machine and after fixing the problem it ends by performing a memory read, so that on return execution can continue as if the call had never been made. However, it will occasionally be the case that there is a genuine page fault which rcache cannot deal with. To handle this case, an address (@3) is passed to rcache in the sequencer counter. The code at this address

will then be called from rcache to restore the state of the machine to how it was before the lnw was started. decpc and noop are two globally known labels provided for use in the common cases where either no modification of the machine state has occurred, or the only modification is the incrementing of the program counter. It is always necessary to push the faulting instruction back into ir0 so that it will be re-executed after the fault has been fixed. noop does this, whilst decpc decrements the pc before falling into noop. It is convenient to use decpc here as the end of the fix up routine.

Five other recaching routines are provided to handle all types of faulting memory references. wcache can be called on a WFLT. It is similar to rcache, but ends with a WR instead of an RD function, whilst rwcache is needed for read/modify/write sequences in which a test must be made for WFLT, but the recache routine must end in a read sequence. Finally, the forms rcacher, wcacher, and rwcacher are similaragain, but end in RRD, RWR, and RRD respectively.

10.7 Byte addressing

Byte addresses are represented in the standard system by using bits 26 and 27 in the address to hold the byte in word bits. This is done because the hardware is word addressed. The OB and OS condition codes are provided to simplify the interpretation of byte addresses. Examples of byte addressing are to be found in the file rv.m which defines a number of indirection operators which take an address from the top of the stack and return the value in the addressed location. Double words and single words are assumed to be aligned on word boundaries, half words on two byte boundaries, and bytes on byte boundaries.

We take as our example rv_c which returns one character from an arbitrary byte address. Referring to the listing we see that the address from the top of the stack is moved straight into the virtual address register so that the memory operation can be started. The OB and OS conditions are used to perform a four way branch to four separate pieces of code each of which extracts the appropriate byte. An additional complication arises from the need to check whether the datum is to be found in the cache memory or in main memory. It will be in main memory if it lies below the cache base register cb, or above cb + 511.

As usual with opcodes which reference memory, if a fault is detected the state of the machine must be restored for the instruction to be retried. A number of further examples of similar instructions is to be found in this file.

10.8 Multiplication and division

Multiplication and division operators for both two's complement and unsigned integers are provided in the files i_b_ari.m, u_b_ari.m and muldiv.m. Of these we have listed here only the first and the last. muldiv.m provides subroutines called from within the other files to perform the inner loop. The division subroutine for two's complement numbers is complicated by unfolding the beginning and the end into

inline code in the calling routine on the dubious grounds of efficiency.

The routines work by using certain special functions provided specifically for the purpose. These special functions modify the ALUSOURCE and ALUFUN fields as well as the carry and register serial input lines as the iteration progresses. We do not attempt to explain their operation in detail here as that requires intimate knowledge of the hardware. If these routines are required for a new instruction set they would best be copied in full and unmodified save for the sections which move the operands and results to and from the stack.

10.9 Default entry points

The file unimp.m demonstrates the use of the DEFAULTENTRY assembler pseudo operator. It is important to ensure when building an instruction set that all entries in the chosen map table are initialized. If not, an incorrect program could result in the decoding of an invalid opcode which would attempt to transfer control via an uninitialized map table entry.

DEFAULTENTRY is similar to ENTRY except that at the time of linking the microcode the linker will automatically initialize all map table entries in the specified map table, which would otherwise be uninitialized, to point to the following microinstruction. In this case a trap is taken passing the unimplemented instruction trap code to the operating system. Note that we need to use a separate DEFAULTENTRY statement for the normal and alternate halves of each map table. Here we choose to point them all at the same two microinstructions.

10.10 Profiling

By making use of multiple instruction sets it is possible to profile the execution of a program dynamically at the individual instruction level. This is particularly useful when developing a new instruction set as it allows the choice of the frequently used special cases to be based upon measurements rather than guesswork. This is illustrated in the file profile.m

This microcode operates in conjunction with the operating system. The code is complicated by the fact that this instruction set is also used to implement single stepping for the benefit of the debuggers. We assume that the register pr, the profiling base register, will be set before the program is run to point to a suitable area of virtual memory. A spare instruction set is loaded entirely by the two default entry points so that all opcodes in the instruction set will result in the execution of one or other of these two pieces of code. The value in IR is read using CAIR and used as an index into the profiling area. The value at that location is incremented.

The H section of IR is now reloaded using the current instruction set field of psw so that the CJV will transfer control to the appropriate entry in the true instruction set of the program. However, before the CJV takes effect the H section is reloaded again. This time the instruction set number of the profiler is restored so that after the next opcode it will regain control and the cycle will be repeated. The profiling is performed at a cost of approximately 1us per opcode which

represents a factor of only about 2 in execution speed in the standard system.

This technique can easily be extended, for example, to form a two dimensional profile of opcode versus byte parameter, or opcode versus following opcode to allow even more optimization of the instruction set.

10.11 Speed control

So far we have not mentioned the speed settings of microinstructions. This is because calculation of the optimum speed of microinstructions requires detailed knowledge of the hardware. The control word contains a two bit field which selects one of four speed settings for each instruction. The mnemonics are

S8	200ns
S7	175ns
S6	150ns
S5	125ns

Most instructions operate at S5, a few at S6, and the remainder at S7 or S8.

The default speed setting in the assembler is S8 which guarantees correct operation if not optimum speed. A program called speed exists which can be applied as a post processor to select the optimum speeds for instructions whose speeds have not been set explicitly. The microprogrammer need never again give thought to the problem.

APPENDIX A

Standard Microassembler Definitions

This appendix provides a source listing of the ms definitions file (msovl1.m) used for the standard system. It serves both as an example of the use of the initialization features of the ms microassembler and as a definitive specification of the control word fields and field values used in the ORION CPU.

```
/*
        msovll.m  --  Standard microcode fields mnemonics and
                        default values.

        Copyright (c) 1982, 1983 by High Level Hardware Limited
*/

*S OFF
*U ON

// Define REGISTER and its printstyle.

        FIELD   REGISTER
        MODESTRING REGISTER     " R%N"

/*

Define the microcode word width, the microcode store length, and the
microcode store page size. The assembler currently only uses the word
width, passing the other parameters direct to the linker.

*/

        WIDTH   64
        LENGTH  32 * 1024
        PAGE    4 * 1024

/*

Do much the same for the entry point store, i.e. the width of the word,
the length of the store, and the page size. All this information is
passed straight to the linker.

*/

        ENTWIDTH 16
        ENTLEN  8 * 1024
        ENTPAGE 256

/*

Set up the parity fields for the microcode and entry point stores. Both
take odd parity computed over the whole word.

*/

        PARITY  44      ODD
        ENTPARITY 15    ODD


// 2901 A register field.

        FIELD   A,      0, 1, 2, 3
```

```
        MODE    A       REGISTER

// 2901 B register field.

        FIELD   B,      4, 5, 6, 7
        MODE    B       REGISTER


// Define symbolic register names.

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

// Set defaults for A and B registers.

        DEFAULT A       R0
        DEFAULT B       R0


// 2901 ALU source field.

        FIELD   ALUSOURCE, 14, 15, 16

/*

The 2901 ALU has two operands R and S.  R is specified by the first
letter and S by the second.  A is the register chosen by the A field, B
is the register chosen by the B field, Q is the Q register, D is the
content of the D bus, and Z is the number 0.

*/

        AQ      =       ALUSOURCE 2     // R=A, S=Q
        AB      =       ALUSOURCE 3     // R=A, S=B
        ZQ      =       ALUSOURCE 0     // R=0, S=Q
        ZB      =       ALUSOURCE 1     // R=0, S=B
        ZA      =       ALUSOURCE 6     // R=0, S=A
        DA      =       ALUSOURCE 7     // R=D, S=A
```

```
        DQ      =       ALUSOURCE 4     // R=D, S=Q
        DZ      =       ALUSOURCE 5     // R=D, S=Z

        DEFAULT ALUSOURCE ZA


// 2901 ALU function field.

        FIELD   ALUFUN, 11, 12, 13

        ADD     =       ALUFUN 1        // R + S, Addition.
        SUBR    =       ALUFUN 0        // S - R, Subtraction.
        SUBS    =       ALUFUN 3        // R - S, Subtraction.
        SUB     =       ALUFUN 3        // SUBS alternative mnemonic.
        OR      =       ALUFUN 2        // R | S, Logical OR.
        AND     =       ALUFUN 5        // R & S, Logical AND.
        NOTRS   =       ALUFUN 4        // R̄ & S, Mask.
        EXOR    =       ALUFUN 7        // R xor S, Exclusive OR.
        EXNOR   =       ALUFUN 6        // R xnor S, Exclusive NOR.

        DEFAULT ALUFUN  OR


// 2901 ALU destination field.

        FIELD   ALUDEST, 8, 9, 10

        NOP     =       ALUDEST 1       // Result ignored.
        QREG    =       ALUDEST 0       // Store in Q.
        RAMA    =       ALUDEST 2       // Store in B, put A on output.
        RAMF    =       ALUDEST 3       // Store in B, put result on output.
        RAMQD   =       ALUDEST 4       // Store in B, shift result and Q down.
        RAMD    =       ALUDEST 5       // Store in B, shift result down.
        RAMQU   =       ALUDEST 6       // Store in B, shift result and Q up.
        RAMU    =       ALUDEST 7       // Store in B, shift result up.

        DEFAULT ALUDEST NOP

// 2901 ALU carry in.

        FIELD   CINX,   17

// Specifying CIN generates a carry in, the default does not.
// Note that in subtraction operations the 2901 carry is inverted.

        CIN     =       CINX 0

        DEFAULT CINX    1

// D bus shifter.

        FIELD   SHIFTER, 18, 19, 20, 23, 24, 25
```

```
/*

Consider the unmodified word to consist of four bytes labelled ABCD with
D being the least significant. The shifter performs a rotation through
0, 1, 2, or 3 bytes and then applies a mask with one bit per byte to
specify which bytes should be passed and which set to zero. The SHIFTER
code represents the resulting word after passing through the shifter,
with Z indicating a byte with the value 0.

 The first set of mnemonics defines all possible operations.

*/
        DABC    =       SHIFTER 0
        CDAB    =       SHIFTER 1
        BCDA    =       SHIFTER 2
        ABCD    =       SHIFTER 3
        ZABC    =       SHIFTER 4
        ZDAB    =       SHIFTER 5
        ZCDA    =       SHIFTER 6
        ZBCD    =       SHIFTER 7
        DZBC    =       SHIFTER 8
        CZAB    =       SHIFTER 9
        BZDA    =       SHIFTER 10
        AZCD    =       SHIFTER 11
        ZZBC    =       SHIFTER 12
        ZZAB    =       SHIFTER 13
        ZZDA    =       SHIFTER 14
        ZZCD    =       SHIFTER 15
        DAZC    =       SHIFTER 16
        CDZB    =       SHIFTER 17
        BCZA    =       SHIFTER 18
        ABZD    =       SHIFTER 19
        ZAZC    =       SHIFTER 20
        ZDZB    =       SHIFTER 21
        ZCZA    =       SHIFTER 22
        ZBZD    =       SHIFTER 23
        DZZC    =       SHIFTER 24
        CZZB    =       SHIFTER 25
        BZZA    =       SHIFTER 26
        AZZD    =       SHIFTER 27
        ZZZC    =       SHIFTER 28
        ZZZB    =       SHIFTER 29
        ZZZA    =       SHIFTER 30
        ZZZD    =       SHIFTER 31
        DABZ    =       SHIFTER 32
        CDAZ    =       SHIFTER 33
        BCDZ    =       SHIFTER 34
        ABCZ    =       SHIFTER 35
        ZABZ    =       SHIFTER 36
        ZDAZ    =       SHIFTER 37
        ZCDZ    =       SHIFTER 38
        ZBCZ    =       SHIFTER 39
        DZBZ    =       SHIFTER 40
```

```
        CZAZ    =       SHIFTER 41
        BZDZ    =       SHIFTER 42
        AZCZ    =       SHIFTER 43
        ZZBZ    =       SHIFTER 44
        ZZAZ    =       SHIFTER 45
        ZZDZ    =       SHIFTER 46
        ZZCZ    =       SHIFTER 47
        DAZZ    =       SHIFTER 48
        CDZZ    =       SHIFTER 49
        BCZZ    =       SHIFTER 50
        ABZZ    =       SHIFTER 51
        ZAZZ    =       SHIFTER 52
        ZDZZ    =       SHIFTER 53
        ZCZZ    =       SHIFTER 54
        ZBZZ    =       SHIFTER 55
        DZZZ    =       SHIFTER 56
        CZZZ    =       SHIFTER 57
        BZZZ    =       SHIFTER 58
        AZZZ    =       SHIFTER 59
        ZZZZ    =       SHIFTER 63

// The following are duplicates with greater mnemonic value.

        RTL0    =       SHIFTER 3       // Rotate left n bytes.
        RTL1    =       SHIFTER 2
        RTL2    =       SHIFTER 1
        RTL3    =       SHIFTER 0

        RTR0    =       SHIFTER 3       // Rotate right n bytes.
        RTR1    =       SHIFTER 0
        RTR2    =       SHIFTER 1
        RTR3    =       SHIFTER 2

        SHR0    =       SHIFTER 3       // Shift right n bytes, bringing in 0.
        SHR1    =       SHIFTER 4
        SHR2    =       SHIFTER 13
        SHR3    =       SHIFTER 30

        SHL0    =       SHIFTER 3       // Shift left n bytes, bringing in 0.
        SHL1    =       SHIFTER 34
        SHL2    =       SHIFTER 49
        SHL3    =       SHIFTER 56

        MASK    =       SHIFTER 31      // Equivalent to  ZZZD

// The default passes the data through unchanged.

        DEFAULT SHIFTER ABCD

/*

2901 ALU serial input control.  This field  controls  the  value  shifted
into  the  ALU  on  a RAMU, RAMD, RAMQU, or RAMQD operation.  Two sets of
```

```
mnemonics are provided because the operation is modified by the  hardware
during certain special functions.

*/

        FIELD   SIN,    21, 22

        ZERO    =       SIN 0           // Shift in 0 (normal set).
        US      =       SIN 0           // Unsigned multiplication &
                                        // division (alternate set).
        ONE     =       SIN 1           // Shift in 1 (normal set).
        DROT    =       SIN 1           // Rotate Q and R as one double
                                        // length register (alternate set).
        ROT     =       SIN 2           // Rotate Q and R independently
                                        // (normal set).
        QRSAVE  =       SIN 2           // Bit saved after QSAVE or RSAVE
                                        // (alternate set).
        ARI     =       SIN 3           // Arithmetic shift (normal set).
        TC      =       SIN 3           // Two's complement multiplication &
                                        // division (alternate set).

// The default provides 'logical' shifts.

        DEFAULT SIN     ZERO

// Speed selection.

        FIELD   SPEED,  26, 27

        S5      =       SPEED 3                 // 125 ns cycle time.
        S6      =       SPEED 2                 // 150 ns.
        S7      =       SPEED 1                 // 175 ns.
        S8      =       SPEED 0                 // 200 ns.

// The default provides safety if not performance.

        DEFAULT SPEED   S8


// The branch field provides 12 bit microcode branch addresses
// and sign extended constants to the ALU.

        FIELD   BRCH,   28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39

        MODE    BRCH    NUMBER

        DEFAULT BRCH    0

// The OPCODE field selects the 2910 sequencer function.

        FIELD   OPCODE, 40, 41, 42, 43

        JZ      =       OPCODE 0        // Jump to 0 (initialization).
```

```
        CJS     =       OPCODE 1        // Conditional call subroutine.
        JMAP    =       OPCODE 2        // Unconditional jump.
        JUMP    =       OPCODE 2        // JMAP alternative mnemonic.
        CJP     =       OPCODE 3        // Conditional jump.
        PUSH    =       OPCODE 4        // Push address for loop control.
        JSRP    =       OPCODE 5        // Conditional call subroutine.
        CJV     =       OPCODE 6        // Instruction dispatch.
        JRP     =       OPCODE 7        // Conditional jump.
        RFCT    =       OPCODE 8        // Loop control.
        RPCT    =       OPCODE 9        // Loop control.
        CRTN    =       OPCODE 10       // Conditional return.
        CJPP    =       OPCODE 11       // Loop termination.
        LDCT    =       OPCODE 12       // Load counter.
        LOOP    =       OPCODE 13       // Loop control.
        CONT    =       OPCODE 14       // Continue sequentially.
        TWB     =       OPCODE 15       // Three way branch.

        DEFAULT OPCODE  CONT

/*

The parity field is filled in by the linker after addresses have been
fixed.  However, for diagnostic purposes a parity error can be generated
on an instruction by setting this bit.

*/

        FIELD   PARITYBIT,      44

        PARITYERROR =   PARITYBIT 1

        DEFAULT PARITYBIT       0               // No error.

/*

2910 condition code selection.  The condition code corresponds to the
result of the current ALU operation.  16 condition codes are available
together with the 16 inverse conditions.

*/

        FIELD   CC,     45, 46, 47, 48, 49

        Z       =       CC 0            // Zero.
        NZ      =       CC 1
        CS      =       CC 2            // Corrected sign.
        NCS     =       CC 3
        C       =       CC 4            // Carry.
        NBW     =       CC 4
        NC      =       CC 5
        BW      =       CC 5            // Borrow.
        S       =       CC 6            // Sign.
        NS      =       CC 7
```

```
        TRUE    =       CC 8            // Guaranteed success.
        T       =       CC 8
        FALSE   =       CC 9            // Guaranteed failure.
        F       =       CC 9
        ODD     =       CC 10           // Odd result on AY bus.
        OD      =       CC 10
        EVEN    =       CC 11           // Even result on AY bus.
        EV      =       CC 11
        LASTCC  =       CC 12           // Condition on previous instruction.
        LC      =       CC 12
        NLASTCC =       CC 13
        NLC     =       CC 13
        O       =       CC 14           // Overflow.
        NO      =       CC 15
        OB      =       CC 16           // Odd byte (AY26 set).
        EB      =       CC 17
        PE      =       CC 18           // Parity error.
        NPE     =       CC 19
        RFLT    =       CC 20           // Read fault in memory.
        NRFLT   =       CC 21
        WFLT    =       CC 22           // Write fault in memory.
        NWFLT   =       CC 23
        INT     =       CC 24           // Interrupt request.
        NINT    =       CC 25
        OS      =       CC 26           // Odd short (AY27 set).
        ES      =       CC 27
        NTBP    =       CC 28
        TBP     =       CC 29           // Translation buffer parity error.
        NMP     =       CC 30
        MP      =       CC 31           // Memory parity error.

// The default ensures that conditional instructions will pass.

        DEFAULT CC      TRUE


// Cache write control.

        FIELD   CWRX,   50

        CWR     =       CWRX 1          // Write to cache.

        DEFAULT CWRX    0               // Don't write.

/*

The following field has several functions. It controls the operation of
the cache address register and the instruction register, and it selects
special functions. Because not all combinations are allowed for this
field the assembler must be told about the allowed combinations. This is
done later. See the CA, IR, and SFUNC fields below.

*/
```

```
        FIELD   CA.IR.SFUNC, 51, 52, 53, 54, 55

// Virtual address register load control.

        FIELD   LVARX,  56

        LVAR    =       LVARX 0         // Load virtual address register.

        DEFAULT LVARX   1               // Don't load.


// Bus register load control.

        FIELD   LBRX,   58

        LBR     =       LBRX 0          // Load bus register.

        DEFAULT LBRX    1               // Don't load.


// D bus source selection.

        FIELD   D,      59, 60, 57

        ALU     =       D 0             // Result of current ALU operation.
        BUS     =       D 1             // Memory bus.
        BR      =       D 2             // Sign extended branch field.
        CSH     =       D 3             // Cache.
        TB      =       D 4             // Translation buffer.
        CAIR    =       D 5             // CA, IR and status bits.
        VAR     =       D 6             // Virtual address register.

        DEFAULT D       ALU

/*

Memory system control.  Note that the MFUNC field specifies the operation
to be performed on the NEXT cycle.

*/

        FIELD   MFUNC,  61, 62, 63

        IDLE    =       MFUNC 0         // No operation, release bus.
        ADDR    =       MFUNC 1         // Send address.
        LRD     =       MFUNC 2         // Read even word.
        RD      =       MFUNC 2         // LRD alternative mnemonic.
        LWR     =       MFUNC 3         // Write even word.
        WR      =       MFUNC 3         // LWR alternative mnemonic.
        LOCK    =       MFUNC 4         // Lock bus during operation.
        RADDR   =       MFUNC 5         // Send address and save in
                                        // input bus register.  Can be
```

```
                                        // read with D=BUS.
        RRD     =       MFUNC 6         // Read odd word.
        RWR     =       MFUNC 7         // Write odd word.

        DEFAULT MFUNC   IDLE


// The CA, IR, and SFUNC fields have no bits assigned to them.
// Instead, the assembler multiplexes them all into CA.IR.SFUNC.

// Cache address register control.

        FIELD   CA

        NOPCA   =       CA 0            // No operation.
        HLDCA   =       CA 5            // Load HIR.
        LDCA    =       CA 4            // Load CA.
        ALDCA   =       CA 1            // Load CA, set ACA.
        INCCA   =       CA 2            // Increment CA.
        DECCA   =       CA 3            // Decrement CA.

        DEFAULT CA      NOPCA


// Instruction register control.

        FIELD   IR

        NOPIR   =       IR 0            // No operation.
        FETCH   =       IR 3            // Set to 1.
        HLDIR   =       IR 4            // Load HIR.
        LDIR    =       IR 1            // Load IR.
        ALDIR   =       IR 5            // Load IR, set AIR.
        PLDIR   =       IR 2            // Load IR from AY8-AY15.

        DEFAULT IR      NOPIR

// Special function selection.

        FIELD   SFUNC

        MUL     =       SFUNC 16        // Multiplication.
        TCDIV   =       SFUNC 17        // Two's complement division.
        DIVL    =       SFUNC 18        // Division, last operation.
        USDIVF  =       SFUNC 19        // Unsigned division, first operation.
        TCDIVF  =       SFUNC 20        // Two's complement division,
                                        // first operation.
        CSAVE   =       SFUNC 21        // Preserve carry out.
        FUN6    =       SFUNC 22        // Unused.
        OUTINT  =       SFUNC 23        // Interrupt Z80, for output.
        USDIV   =       SFUNC 24        // Unsigned division.
        ASIN    =       SFUNC 25        // Alternate SIN select.
        RSAVE   =       SFUNC 26        // Save bit shifted out of RAM.
```

```
        QSAVE   =       SFUNC 27        // Save bit shifted out of QREG.
        ININT   =       SFUNC 28        // Interrupt Z80, for input.
        WRMM    =       SFUNC 29        // Write memory mode.
        TBWR    =       SFUNC 30        // Write to translation buffer.
        CLRPERR =       SFUNC 31        // Clear parity errors.


// Special flag to assembler.

        DEFAULT SFUNC   -1

// Signal to assembler that the CA, IR, and SFUNC fields should be
// combined.

        MULTIPLEX CA.IR.SFUNC 0 NOPCA NOPIR SFUNC -1
        MULTIPLEX CA.IR.SFUNC 1 NOPCA LDIR  SFUNC -1
        MULTIPLEX CA.IR.SFUNC 2 NOPCA PLDIR SFUNC -1
        MULTIPLEX CA.IR.SFUNC 3 NOPCA FETCH SFUNC -1
        MULTIPLEX CA.IR.SFUNC 4 LDCA  NOPIR SFUNC -1
        MULTIPLEX CA.IR.SFUNC 5 LDCA  FETCH SFUNC -1
        MULTIPLEX CA.IR.SFUNC 6 INCCA NOPIR SFUNC -1
        MULTIPLEX CA.IR.SFUNC 7 INCCA LDIR  SFUNC -1
        MULTIPLEX CA.IR.SFUNC 8 INCCA PLDIR SFUNC -1
        MULTIPLEX CA.IR.SFUNC 9 NOPCA HLDIR SFUNC -1
        MULTIPLEX CA.IR.SFUNC 10 DECCA NOPIR SFUNC -1
        MULTIPLEX CA.IR.SFUNC 11 DECCA LDIR  SFUNC -1
        MULTIPLEX CA.IR.SFUNC 12 DECCA PLDIR SFUNC -1
        MULTIPLEX CA.IR.SFUNC 13 ALDCA NOPIR SFUNC -1
        MULTIPLEX CA.IR.SFUNC 14 HLDCA NOPIR SFUNC -1
        MULTIPLEX CA.IR.SFUNC 15 NOPCA ALDIR SFUNC -1
        MULTIPLEX CA.IR.SFUNC 16 NOPCA NOPIR SFUNC 16
        MULTIPLEX CA.IR.SFUNC 17 NOPCA NOPIR SFUNC 17
        MULTIPLEX CA.IR.SFUNC 18 NOPCA NOPIR SFUNC 18
        MULTIPLEX CA.IR.SFUNC 19 NOPCA NOPIR SFUNC 19
        MULTIPLEX CA.IR.SFUNC 20 NOPCA NOPIR SFUNC 20
        MULTIPLEX CA.IR.SFUNC 21 NOPCA NOPIR SFUNC 21
        MULTIPLEX CA.IR.SFUNC 22 NOPCA NOPIR SFUNC 22
        MULTIPLEX CA.IR.SFUNC 23 NOPCA NOPIR SFUNC 23
        MULTIPLEX CA.IR.SFUNC 24 NOPCA NOPIR SFUNC 24
        MULTIPLEX CA.IR.SFUNC 25 NOPCA NOPIR SFUNC 25
        MULTIPLEX CA.IR.SFUNC 26 NOPCA NOPIR SFUNC 26
        MULTIPLEX CA.IR.SFUNC 27 NOPCA NOPIR SFUNC 27
        MULTIPLEX CA.IR.SFUNC 28 NOPCA NOPIR SFUNC 28
        MULTIPLEX CA.IR.SFUNC 29 NOPCA NOPIR SFUNC 29
        MULTIPLEX CA.IR.SFUNC 30 NOPCA NOPIR SFUNC 30
        MULTIPLEX CA.IR.SFUNC 31 NOPCA NOPIR SFUNC 31

*U OFF
*S ON
```

APPENDIX B

Ms Microassembler Users Manual

Ms is a microassembler for ORION microcode. It converts source text into an intermediate relocatable binary file which can be linked with a number of other such files using the microlinker ml. Libraries of relocatable binary files may be maintained with the microlibrarian mb. Amongst the facilities offered are conditional assembly, relocation, external and global symbols, and cross-reference lists.

Ms is a general purpose microassembler that reads a description of the microcode syntax from a definitions file prior to assembly. The general syntax of a microinstruction is described informally in Chapter 2. Appendix A lists the definitions file for the standard system.

B.1 Assembler operation

The general form of invoking ms at OTS command level is

ms options file

where options represents any number of option arguments and file is the name of the microcode source file to be assembled. The assembler normally reads the source from file.m and outputs relocatable binary to file.mrl; the default extension of the source file name (.m) can be overridden by specifying a non-blank extension. The options, which may appear in any order, are

-o outfile
: The relocatable output is sent to outfile, with a default extension of .mrl.

-l listfile
: An annotated listing is sent to listfile. If the -l option appears at the end of the command line with no listfile specified, the annotated listing is sent to file.prn (or to outfile.prn if a -o switch is given).

-c
: The annotated listing is sent to the user's terminal; the -l and -c options are mutually exclusive. In either case, the listing includes a printout of the object code and the addresses of the instructions, error messages, and a copy of the source file. A symbol table appears at the end.

-i initfile
: By default, definitions of the microcode syntax are read from the file msovl1.m which is reproduced in Appendix A. Option -i causes the definitions to be read from initfile (with a default extension of .m). This may be used for debugging or to configure the assembler for different hardware.

-n Disables line numbering in the listing file. The listing file normally includes a column containing line numbers. Setting the -n switch suppresses these numbers. (This switch only has effect if one of the switches -c or -l is causing a listing file to be generated.) Including a *NUMBERS OFF directive in the source file is equivalent.

-u Allows uppercase only identifiers to be used. Normally, a user defined identifier should contain at least one lower case letter. Setting the -u switch, which is equivalent to issuing a *UPPERCASE ON directive, allows such identifiers.

Multiple option letters (except for -o, -i and -l) may follow a single minus sign character.

The remainder of this manual is devoted to a description of the syntax of microcode source files.

B.2 Elements of assembly language

A microprogram consists of a series of lines of text called statements. Within each statement the characters are grouped to form symbols, operators, and numbers. The general form of a statement is

```
label:   microinstruction          // comment
```

where label is a user defined symbol, microinstruction is a composite element made up from a combination of symbols, operators and numbers, and comment is an arbitrary sequence of characters, preceded by two slash characters. Any of these components may be absent, and indeed a completely blank line is a valid statement. However, a label, if present, must come first, and a single line comment (if any) last.

B.2.1 Symbols

Symbolic names provide a means by which to refer to various entities in a readable fashion. Symbols fall into two main types; those built into ms or defined by the definitions file, and those created by the programmer. The built in symbols include the names of the fields in the control word and the mnemonics corresponding to the possible values which may be taken on by those fields; the symbols in this group are referred to as microinstruction field names and microinstruction field values. Another group of built in symbols is called pseudo-operators, that is, symbols that are used like machine instructions but whose purpose is to control the assembler. A subset of these (e.g. FIELD, MODE, DEFAULT and MULTIPLEX) is of particular importance since it provides the means by which the microinstruction fields are defined.

User defined symbols are used as program labels to identify points in the code which are the targets of control transfer instructions and as mnemonic names for numeric values.

Symbols are composed of a group of one or more characters, starting with a letter (A to Z, a to z) or a dot (.), underscore (_), or commercial at sign (@), and continuing with any of these characters or with decimal digits (0 to 9). Symbols may be any reasonable length (up to a limit of 255 characters).

To reduce the chance of duplication between the rather large number of predefined symbols (microinstruction field names, field values and pseudo-ops) and user defined symbols, a naming convention is normally enforced. No predefined symbol contains a lower case letter, whilst user defined names must contain at least one lower case letter or commercial at sign. Breaches of this rule will cause an L error unless it has been disabled with a *U ON directive or a -u option on the command line. This escape mechanism is provided to allow additional built in symbols to be defined and is used, for example, in the definitions file.

B.2.2 Labels

A label is a special case of a user defined symbol in which the symbol becomes defined by its appearance at the beginning of a statement and followed by a colon; both a value and a relocation type are assigned. The value assigned to the label is the address of the next microinstruction. The relocation type is that of the current program counter. A label may be preceded in a statement only by another label, and any type of statement may be labelled although it is common practice only to label instructions and blank lines.

Labels that begin with a letter remain in scope for the whole of the assembly and will be included in the symbol table. Labels that begin with a commercial at sign (@) are local and are in scope only between the surrounding pair of ordinary labels. It is allowable to define multiple local labels of the same name, provided that their definitions are separated by the definition of at least one non-local label. Local labels will not appear in the symbol table.

All labels must be unique within a single assembly (or, in the case of local labels, within one scope block), and must be distinct from other built in or user defined symbols, otherwise a phase (P) error will result. A phase error will also be generated if the value of a label on pass 1 differs from that during pass 2. The most likely way in which this condition may arise is through incorrect usage of conditional assembly statements.

B.2.3 Symbol definitions

Definitions provide symbolic names for numeric and other values which can be used later when the names are encountered in other statements. A typical example of a symbol definition might be

```
foo     =       37
```

which gives the value 37 to the symbol foo. The purpose of definitions is to increase readability and localize the occurrence of 'magic numbers'. Symbols must be defined before they are used. However, ms is a two pass assembler and it is usually adequate to ensure that all symbols are defined by the end of the first pass. See also the descriptions of the pseudo-ops SET and EQU, below.

B.2.4 Numbers

A number is a string of digits in binary, octal, decimal, or hexadecimal, possibly terminated by a trailing radix indicator. Valid indicators are:

B	Binary
O or Q	Octal
D or .	Decimal
H	Hexadecimal

In the absence of a trailing indicator, the default radix is used. The default radix is initially decimal; the pseudo-operators RADB, RADO, or RADH can be used to set the default radix to 2, 8, or 16 respectively. Hexadecimal numbers must start with a decimal digit (a zero will suffice), and the letters A to F (or a to f) are used for the digits greater than 9.

The permitted size of a number is a machine-dependent quantity, being limited by the size of a word in the implementation language (BCPL); the version that runs on the ORION hardware uses signed 32 bit integers. The two main uses of numbers in the standard microcode (the branch field and the instruction lookup table) both require relatively small numbers. No overflow checking is performed except for division by zero.

B.2.5 String and character constants

A string consists of a series of characters enclosed in double quotation marks (") and can be used instead of an ordinary identifier. However the main uses of strings are as file and other names, or for prompt strings, in some of the pseudo-operators.

A character constant consists of a single character, possibly preceded by an asterisk, enclosed in single quotation marks (') and is semantically equivalent to a number.

In both strings and character constants the asterisk character (*) is treated as an escape character and in conjunction with the next character generates one of the following special characters

*B	Backspace
*C	Carriage return
*N	Newline
*P	New page (form feed)
*S	Space
*T	Horizontal tab
**	*
*"	"
*'	'

Newlines and double quotes may not be embedded within strings unless escaped.

B.2.6 Expressions

An expression is formed from operands and operators. Operands can be numbers, labels, user defined symbols or built in constants. The only operators available work on numeric quantities.

The available monadic operators are +, -, and \; + does nothing, while - and \ cause negation and bitwise complementation respectively. There may be no more than one monadic operator applied to any one primary. The dyadic operators, in decreasing order of precedence, are:

* / REM	Multiplication, division, remainder
+ -	Addition, subtraction
<< >>	Shift left, shift right
== \= < > <= >=	Equal, not equal, less than, greater than, less than or equal, greater than or equal
↑	Bitwise exclusive or
&	Bitwise logical and
|	Bitwise logical or

Various restrictions apply to arithmetic involving relocatable and external quantities (described below). At least one of the operands of addition must be absolute, and the relocation type of the result is the type of the other argument. The relocation types of the operands of subtraction must either be the same, in which case the type of the result is absolute, or the subtrahend (right operand) must be absolute, in which case the type of the result is that of the first argument. Further, two external quantities may be compared only if they reference the same symbol.

Field names (described below) can be applied to absolute numeric quantities as prefix monadic operators to generate arbitrary values of any desired field type. (This is analogous to casts in C.) For example, in the context of the standard definitions file "MFUNC 3" will generate the memory function whose internal representation is 3. This feature is intended mainly as a way of initializing the built-in symbols in the definitions file.

B.2.7 Layout

A statement normally occupies a single line and is terminated by a newline character. Some instructions can get uncomfortably long, however, and in deference to this the combination 'backslash newline' is treated (except for listing purposes) as white space. There may be any number of space or tab characters between the backslash and the newline.

The semicolon (;) character and the two-character sequence // each introduce a single line comment which lasts until the end of the line. Such a comment is completely ignored except for listing purposes. These comments may not be continued with a trailing backslash.

The character pair /* introduces an extended comment which lasts until the character pair */. Likewise (for historical reasons) the percent (%) character introduces an extended comment which lasts until a

matching percent character. An extended comment may last for less than one line, or it may cover several lines. Extended comments cannot be nested, except that a comment introduced by a percent may contain an embedded comment introduced by /*, and vice versa.

White space (i.e. spaces, tab characters, single-line comments, and extended comments) is generally ignored, except that otherwise contiguous names must be separated by white space. Blank lines are ignored.

B.3 Microinstructions

A microinstruction consists of one or more field assignments. A field assignment takes one of the forms

fieldname = fieldvalue

or simply

fieldvalue

Fieldname is the name of a microcode field, as defined by a FIELD pseudo-op, while fieldvalue is a value of the type required by field fieldname. Field assignments may be separated by a comma or only by spaces and tabs, according to taste. However, there is a parsing ambiguity involving monadic operators which is most easily resolved by the inclusion of a comma: if a field starts with a monadic operator then it will be taken as a dyadic operator belonging to the previous field unless that field was terminated by a comma. Because the fields are defined in the standard definitions file, and because they are very closely connected with the hardware, the reader is directed to Appendix A for examples.

B.4 Pseudo-operators

As noted earlier, pseudo-operators are built in symbols resembling instructions that are used to initialize and control the progress of an assembly. They are divided here into logical groups, each group performing a related set of functions. The first two groups comprise those pseudo-ops used to define and assemble microinstructions. Their descriptions should be read in conjunction with the listing of the standard definitions file in Appendix A.

The remaining groups are associated with the traditional form of assembler control.

B.4.1 Definition of hardware

The pseudo-ops in this group define the shape and size of the control store and map tables.

WIDTH width
: Sets the width of the microcode store to be width bits. width must be absolute and defaults to 64 if a WIDTH is not given.

LENGTH length
: Defines the microcode store to be length words long. length must be absolute. This information is not used by the assembler, but

is passed on to the linker.

PAGE size
Defines the microcode store to be organized into pages of size words. size must be absolute. Again, this information is not used by the assembler but is passed on to the linker.

PARITY parity-bit ODD/EVEN [bit-list]
Defines the microcode bit parity-bit to be a parity bit. parity-bit must be absolute. The parity bit is of odd or even parity depending on whether the following word is ODD or EVEN. The parity is computed over the bits in bit-list, or over all the bits in the microcode word by default if no bit-list is given. There can be more than one parity bit in a word; however the standard system uses only one.

ENTWIDTH width
Defines map table entries to be width bits wide. width must be absolute and must be less than or equal to the size of a word in the implementation language (currently 32 bits). This information is not used by the assembler but is passed to the linker.

ENTLEN length
Defines the total size of the map tables to be length words. length must be absolute. This information is not used by the assembler but is passed to the linker.

ENTPAGE size
Defines the length of each map table to be size words. size must be absolute. This information is not used by the assembler but is passed to the linker.

ENTPARITY parity-bit ODD/EVEN [bit-list]
Defines the map table bit parity-bit to be a parity bit. parity-bit must be absolute. It is of odd or even parity depending on whether the following word is ODD or EVEN. Parity is computed over the bits in bit-list (which is a comma- or space-separated list of absolute bit numbers), or over all the bits in a map table entry by default if no bit list is given. There can be more than one parity bit in a map table entry. The standard system uses one.

B.4.2 Definition of microinstructions

This group of pseudo-ops defines the microinstruction field names and field values.

FIELD name [,e1, e2 ...]
Defines a new microcode field, henceforth referred to as name. The absolute expressions e1, e2, ... specify the bit numbers which the field will occupy, least significant first. These numbers should be in the range zero to width - 1. name may not be a local symbol.

After a field name has been specified using the FIELD pseudo-op, field values can be defined using the EQU operator or equals sign (see below). The value assigned is converted to the appropriate type by applying the field name as a monadic operator. For

example

```
FIELD ALUSOURCE, 14, 15, 16
AQ = ALUSOURCE 2
```

The second statement associates both a type (ALUSOURCE) and a value (2) to the symbol AQ. This then allows the assembler to check that a variable that is being assigned to a field is of an appropriate type, as in

```
ALUSOURCE = AQ
```

but more importantly allows the short form of field assignment (where only the field value is specified) to be used.

Note that it is meaningful to define a dummy field with no bits assigned to it. This occurs either when the same set of field values can be assigned to more than one field (e.g. in the case of the A and B register fields in the standard system), or when several logical fields are to be multiplexed into one hardware field.

MODE field type

Sets the field type of field to be type. type is the name of a previously defined field name or the built-in name NUMBER. The latter is useful where a field (for example the branch field in the standard system) takes on a numeric value, or where the same set of field values are to be used for more than one field. For example

```
FIELD REGISTER
R0 = REGISTER 0
FIELD A, 0, 1, 2, 3
MODE A REGISTER
FIELD B, 4, 5, 6, 7
MODE B REGISTER
```

after which both 'A = R0' and 'B = R0' are valid field assignments.

MODESTRING field string

Sets the print style of values of type field to be string. field is the name of a previously defined field. string is a string which controls the print style of values of the specified type in the symbol table. Most characters from string will be printed literally. The percent character (%) introduces a value field which is replaced by the value of the symbol according to the table below. n represents a decimal digit.

%Bn	Binary, n digits wide
%In	Decimal, n digits wide
%N	Decimal, minimum width
%On	Octal, n digits wide
%Xn	Hexadecimal, n digits wide

The sequence %% represents a single %. The resulting string will be truncated or padded on the right with spaces to eight characters.

DEFAULT name value
Sets the default value for the microcode field specified by name (which must be a valid microcode field name) to value (which may be absolute or of the same type as the field). The default value of a field will be used automatically in an instruction if no value has been assigned explicitly. There are no checks on the range of the default, but an attempt to force a value into too few bits will fail later with a V error.

MULTIPLEX dest, s1[, s2, ... sn]
Used to define fields which cannot be represented by a single name. Wherever the combination s1, s2, ... sn appears in an instruction (where the si are simple field values), dest will be inserted into the instruction. dest must be a field value.

There are occasions when several logically independent fields are actually controlled by a single (vertical) field in the control word which is decoded by the hardware at run time. MULTIPLEX allows the microprogammer to think in terms of the logical fields whilst the assembler takes care of combining them into the single physical field. See the standard definitions file for an example.

DEFAULTINSTR instruction
Sets up a default microinstruction which the linker will use to fill all otherwise uninitialized microcode words in the current page. The instruction following the pseudo-operator uses the normal microinstruction syntax. There is no known use for this facility except in diagnostic microcode.

ENTRY num
Makes an entry in the map table. The numth entry in the table is set to the address of the next microinstruction. num must be absolute. If the last primary in the expression which evaluates to num is a name, then this name is transmitted to the linker for use in making useful load maps.

DEFAULTENTRY entry
Causes the page in the instruction lookup table whose lowest address is entry to be initialized so that all unused entries are set to the address of the next microinstruction. entry must be a multiple of the page size, although this is only enforced by the linker. This provides a convenient way to trap undefined opcodes (i.e. map table entries).

B.4.3 Program counter control

ASEG
Selects the absolute segment. The type of the program counter is set to absolute and its value to whatever it was last time ASEG was in effect, or zero initially. ASEG has no effect when already in ASEG. Instructions assembled into ASEG will not be relocated by the linker. ASEG is mainly useful in diagnostic microcode.

CSEG
Selects the relocatable segment. The type of the program counter is set to relocatable and its value to whatever it was last time

CSEG was in effect, or zero initially. CSEG has no effect when already in CSEG. CSEG is entered by default at the start of each pass.

DEFS size
Increases the program counter by size, which must be absolute. The only known use of DEFS is in a trick to generate a parity error in a map table, by means of the sequence

```
DEFS    32768
ENTRY   dummy     ; Where dummy is the desired entry point
DEFS    -32768
```

In the standard system this works because the parity bit is stored in bit 15 of the map table. Parity computed by the linker is exclusive ORed into the defined parity bit. Thus, setting the parity bit with the assembler causes the linker to generate a parity error.

ORG address
Sets the program counter to be address, which must be either absolute or the same relocatable type as the program counter. This pseudo-operator is used most often in ASEG, although it will work in CSEG.

END [start]
Signals the end of the program segment; start, if given, sets the start address. start may be of any numeric type. (No use for the start address is currently known.)

B.4.4 Relocation

Microcode programs fall naturally into many small and relatively independent sections and it is very convenient to be able to work on these individually. In order to support this, the microassembler produces relocatable output which is independent of the final address it will occupy in the control store. The relocatable output file from an assembly is called a module. Modules are combined by the microlinker ml, described in Appendix C, to form a memory image file that can be loaded into the control store.

EXTERNAL symbol [,symbol ...]
Makes the specified symbols external, i.e. defined in some other module. None of these symbols may be already defined in any way. These symbols are given values at link time by specifying them as GLOBAL in the other module. symbol may not be a local symbol.

FORCE symbol [,symbol ...]
Similar to EXTERNAL except that it forces a reference to each of the specified symbols. The linker will ensure that each of these symbols is defined, even if there is no other explicit reference it. The main use of FORCE is to avoid dummy references to external symbols in order to cause modules to be loaded from a library. This is necessary because control can be passed to a microinstruction via a map table entry without that instruction being labelled explicitly. symbol may not be a local symbol.

GLOBAL symbol [,symbol ...]
Sets the specified symbols to be global, i.e. available as externals in other modules. Global symbols must be given some absolute or relocatable value in the current module. symbol may not be a local symbol.

LIBRARY
Specifies the names of one or more libraries which should be searched by the linker in an attempt to satisfy unresolved EXTERNAL references when all explicitly named modules have been linked. The filenames are in the usual format, with a default extension defined by the linker; it is currently .mrl. The libraries will be searched in the specified order, after any explicit library requests have been processed.

NAME name
Sets the name of the module to be name. The default name is the primary name of the file being assembled. If more than one NAME is given within a single module, the last one given during pass 1 is used. The module name is used by the microlibrarian mb. Beware that confusion can arise if module names are different from filenames.

B.4.5 Conditional assembly

COND Boolean
Starts a conditional assembly block. The Boolean expression following the pseudo-operator is evaluated and if false (zero), assembly is suspended. COND may be nested, up to a maximum of 10 levels.

ELSE
Toggles the current level of conditional assembly. There may be no more than one ELSE between each COND/ENDC pair, and it is illegal to give an ELSE unless there has been a preceding COND.

ENDC
Terminates a conditional assembly block. It is illegal unless there is an unterminated COND in operation. ENDC will normally cause assembly to resume, except in the case of a COND/ENDC nested within the false branch of another COND.

B.4.6 Miscellaneous

EQU
Used in the form

```
name    EQU expr
```

and sets name to have the value and type of expr. expr may be of any type. name must not already be defined except where the new value and type are the same as the previous values. An equals sign (=) may be used as a synonym for EQU.

SET Used in the form

```
name    SET value
```

and similar to EQU, except that it is not an error to redefine a symbol previously defined with SET. A colon followed by an equals sign (:=) may be used as a synonym for SET.

QUERY
Used in the form

name1 QUERY name2

and is similar to EQU except that the value assigned to name1 is taken from the user's keyboard. name2 is used as a prompt; it will often be a quoted string. The response to the QUERY may be any well-formed expression, terminated by a carriage return character. If the expression contains a detectable error, the message

Bad input

is printed and the prompt repeated for another attempt. If name1 is already defined, the QUERY statement is skipped.

RADD
Sets the default radix to decimal. Numbers are treated as decimal unless explicitly flagged with a trailing indicator. The default radix is decimal initially. Experience suggests that it should not be changed.

RADB
Sets the default radix to be binary. Any number which is not explicitly set to some radix by a trailing indicator is treated as a binary number.

RADO
Sets the default radix to be octal. Any number which is not explicitly set to some radix by a trailing indicator is treated as an octal number.

RADH
Sets the default radix to be hexadecimal. Any number which is not explicitly set to some radix by a trailing indicator is treated as a hexadecimal number. Caution should be employed when using RADH, as the trailing indicator supercedes the default radix, so that such numbers as 0D and 1B will be treated as zero (decimal) and one (binary) respectively.

B.5 Directives

An assembler directive is indicated by the occurrence of an asterisk (*) as the first character of a statement, followed by a command word; only the first letter of the command word is significant, the rest of it being skipped. Arguments, if any, are separated from the command by one or more spaces or tabs.

*EJECT
Causes a form feed character to be sent to the listing file.

*HEXLIST
Takes a single ON or OFF argument which determines whether the address fields of the listing (instruction addresses and map table data) are in hexadecimal or octal. Multiple *H directives nest.

Address fields are initially listed in hexadecimal.

*INCLUDE file
Causes the source of file to be included in the assembly at that point. file is of the usual format, with a default extension of .m. Includes nest arbitrarily deep. The listing level is decreased by one for each depth of include, so that included files are not listed. The user defined symbols flag is also decremented by one for each include, so that by default symbols defined within included files are not included in the symbol table. Line numbers, if enabled, start again (at 1) for each included file.

*LISTING
Takes a single ON or OFF argument and increases or decreases the listing level by one. Multiple *L directives nest. Listing is initially on if a listing has been requested on the command line. A *L ON directive is required before a *I if the contents of the included file are to appear in the listing.

*NUMBERING
Takes a single ON or OFF argument and enables or disables line numbering. Each included file is numbered separately, and lines are counted even if numbering is temporarily suspended. Line numbering is initially on, unless there is a -n in the command line. Multiple *N directives nest.

*SYMBOLS
Takes a single ON or OFF argument and sets or resets the symbols flag. If this flag is on (the default), any definition (or redefinition with SET) of a symbol will cause it to appear in the symbol table. Otherwise, it will only appear if the symbol includes a lower case letter or digit and there is at least one reference to the symbol.

*UPPERCASE
Takes a single ON or OFF argument and sets or resets the uppercase flag. User defined identifiers which do not contain a lower case letter are not recognised, in order to reduce the risk of clashes with built in symbols. Such identifiers are flagged with an L error unless the uppercase flag is set. This flag is initially reset unless a -u switch is present in the command line. Multiple *U directives nest.

*WIDTH width
Sets the logical width of the symbol table to be width characters. width is an absolute expression. The symbol table will not output characters beyond this limit except in the extreme case where the width is less than the number of characters before the first reference. (The WIDTH directive should not be confused with the WIDTH pseudo operator.)

B.6 Errors

When ms finds an error it flags the output with an error code in column one and a greater than sign in column two of the listing file. The first error report for each file involved in the assembly includes the name of

the relevant file. The erroneous line is echoed to the user's terminal unless the listing file is being sent to the terminal. An up arrow is printed on the next line at the point in the line where the error was detected. The error codes are:

A Expression too complex. The expression stack is ten levels deep. If this error occurs, re-order the expression in a less complex manner.

B Brackets (parentheses) mismatch. Somewhere in an expression there is a left parenthesis without a corresponding right parenthesis.

C Conditional assembly error. This error is generated by an illegal COND, ELSE, or ENDC pseudo-operator. The causes are: COND nested to more than ten levels, ELSE with no COND, ENDC with no COND, two ELSEs after one COND.

D Directive error. This error occurs in the event of an illegal directive.

E Illegal combination of types within an expression. This error occurs after an attempt to perform an arithmetic operation on non-numeric operands or the wrong sort of operation on relocatable or external operands. It can also appear as a result of attempting to use a monadic operator at the start of field separated from preceding fields only by spaces or tabs.

F FIELD or DEFAULT pseudo-operator fault. An attempt to specify a microcode field or default value failed. Failure to define a default correctly will normally only produce incorrect code, but errors in a FIELD pseudo-operator will cause many subsequent errors.

I Illegal character. This error indicates that there are further characters remaining on a line after a statement has been assembled or that a character which cannot form part of any statement has been encountered. In practice it is produced after many typographical errors.

L Identifier all upper case. This error occurs after an attempt to define an identifier (with EQU or SET, or as a label, or with GLOBAL or EXTERNAL) which does not contain at least one lower case letter or digit. This error can be suppressed by the *U ON directive or by -u in the command line. In any event, this error is only a warning.

M Multiply defined symbol. It is illegal to redefine a symbol with EQU unless the new value and type are the same as the old.

N Number error. A number contains a digit too large for the current radix. This error is also generated by numbers containing dots and commercial at signs etc.

P Phase error. This error occurs when the value of a label in pass 2 is not the same as it was in pass 1. In practice, this error usually occurs as a result of the number of instructions being miscalculated on one of the passes, possibly as a result of conditional

assembly problems.

Q Quote error. There is an error in a quoted character or string.

R Multiplexed field clash or microcode field redefined. This error is produced for invalid combinations of field values for a multiplexed field. It is also illegal to attempt to define any microcode field more than once in any instruction. With the standard definitions file the most likely causes are an attempt to use the branch field for both an address and a constant or an invalid combination of IR and CA functions.

S Syntax error. This error is generated when the syntax of a line is wrong, or the type of a statement is unrecognized. This error normally catches the typographical errors which the I error misses.

T Type mismatch. This error occurs when an expression of a specific type is required and one of some other type is supplied. In practice, this occurs after field assignments where the field name is supplied explicitly, or after some of the pseudo-ops which require a numeric or absolute value.

U Undefined symbol. An encountered symbol has not been defined, nor is it built-in.

V Value error. Either attempted division by zero or there are some bits left over after filling in a field in the output with the specified value. In the standard system this usually means that the number specified for the branch field is too large. It may indicate an inconsistency in the definitions file.

Some errors generate other types of warning message.

END statements within included files generate the message

Illegal END in file FILE

where FILE is the name of the offending file.

The message

Bad input

is produced after an illegal expression has been entered after a QUERY pseudo-operator and before the prompt is repeated.

The message

End of file in extended comment

is produced when an extended comment (after % or /*) is still in effect when the end of a file is encountered. The comment is closed.

A few fatal errors are possible. These print out the name and version of the assembler, followed by some other information intended to describe the nature of the error. The assembler aborts.

The message

Can't open FILE

(where FILE is replaced by the offending file name) may be printed at

the start of assembly and at each include statement, and implies an error in opening the file.

Can't create FILE

(where FILE is replaced by the offending file name) may be printed at the start of assembly, and implies a fault in attempting to create the output or listing files.

The reminder

Usage: ms [-o file] [-i initfile] [-l listfile] file [-cnux]

is output when the input line cannot be parsed.

APPENDIX C

Ml Microlinker Users Manual

Ml is a linker for ORION microcode. It accepts as input the relocatable binary files produced by the microassembler ms or the microlibrarian mb and produces an output file in an absolute hexadecimal format suitable for loading into the control store.

The microlinker deals with all the relocation features generated by the microassembler, including references to external symbols. It will search libraries, indicated either by LIBRARY statements within the microassembly source or by appropriate command line arguments, if, after loading all the specified files, there remain undefined global symbols.

The code relocatable segment starts at a fixed location, although this can be moved by an appropriate link time option. Checks are made to ensure that the control store is not overlayed, and warning messages are normally issued if this is attempted. Similar warnings result from attempts to overlay the map tables unless suppressed.

By specifying the appropriate command line arguments, ml will output a load map, giving the load addresses of the various modules, the values of global symbols defined within each module and the addresses of the entry points.

C.1 Linker operation

The general form of invoking ml at OTS command level is

ml options files

where options represents any number of option arguments and files are the names of one or more relocatable files to be loaded. The input files will be loaded in the order specified in the command line and have a default extension of .mrl. The output file name is normally the same as that of the first input file with an extension of .mcd. Most option arguments may appear anywhere on the command line. The options are

-o outfile

Output the linked image to outfile, with a default extension of .mcd. If no output file is specified the output is sent to a file with the same primary name as the first input file but with an extension of .mcd. It is an error to attempt to specify more than one output file name.

-l mapfile

Output a load map to mapfile, with a default extension of .map. No load map is generated in the absence of this option and no more than one load map file name may be specified. If the -l option is the last argument on the command line (i.e. there is no mapfile specified) the load map is sent to file.map, where file is

the primary name of the output file. The load map includes the control store addresses of all defined modules, the values of global variables. If the -a, -e, or -n options are specified the load map will also include the addresses pointed to by the map table entries. A load map will be produced automatically if any of the options -a, -e, or -n are specified.

-a Output to the map file a list of the entry points defined during linking, sorted by control store address as the primary key and entry number as the secondary key. (An entry point is an address in the control store that is pointed to by a map table entry.)

-e Output to the map file a list of the entry points defined during linking, sorted by entry number.

-n Output to the map file a list of the entry points defined during linking, sorted alphabetically by entry name. Unnamed entry points precede all named entry points and are sorted by entry number.

-s libfile
After all input files have been read search the library file libfile to resolve any undefined global symbols. The default extension of libfile is .mrl. The microlinker searches libraries specified by -s options first, in the order given on the command line. Then it searches libraries specified within the input files (by LIBRARY requests in the assembler source) in reverse alphabetical order of their file names. Finally, it searches libraries specified by -z options. Any number of library files may be specified.

-z libfile
Search libfile after all other library searches. See the description of -s above for details.

-cn Start the code relocatable segment at address n, which is in hexadecimal. The default is 0.

-i Suppress warning messages concerning the overlaying of the map tables. Unless this option is given an attempt to redefine an entry point will generate a warning message.

-k Keep the output file, even if errors occur. Normally the output file is deleted if errors occur in order to prevent inadvertent use of incomplete programs.

-m Suppress warning messages concerning the overlaying of the control store. Unless this option is given an attempt to redefine a control store address (in ASEG) generate warning messages.

C.2 Errors

Error conditions are reported by outputting a message to the user's terminal. The first error message is preceded by a line including the name and version number of the linker. Errors are divided into two categories, fatal and non-fatal. ml attempts to continue after non-fatal errors, although by default no output file is created if any errors occur. Fatal errors cause the immediate termination of the linking

process.

C.2.1 Fatal errors

Usage: ml [-o outfile] [-l mapfile] [-sz libfile] [-acNeikmn] files
: Indicates some sort of error condition in the command line, normally either no arguments at all or an illegal option.

No input files on command line
: The command line consists entirely of options and output, load map, and library file specifications.

Can't create filename
: An attempt to create the file filename failed.

Illegal record type n
Illegal record length
: A record of an illegal type or length has been found in an input file or a library. Either the file is corrupt or is not a relocatable binary file (or there is an internal error in the assembler or linker).

Bad DEFGLOBAL type n
: If you get this error, there is a coding error in the linker.

Symbol table overflow
: ml has run out of symbol table space. Short of recompiling the linker to use more space, the only option is to reorder the input files in order to reduce the number of forward references to external symbols.

Errors detected - output file not created
: This error message is produced after the linking process has generated one or more non-fatal errors and the -k option is not set. No output file is created.

Trailing output request
: The command line ends with a -o option.

C.2.2 Non-fatal errors

Trailing library request
: The command line ends with one of the options -s or -z. The option is ignored.

Can't open filename
: An attempt to open the file filename failed.

Undefined symbols:
: This message is followed by a list of symbols still undefined after all the specified libraries have been searched.

Control store address n too large
Entry address n too large
Entry point n (name) too large
: There has been an overflow in either the control store in the first case or the map tables in the other two. The first normally means that there is more code than will fit since the control store is usually filled sequentially.

Address n already loaded
Entry n already defined
: An attempt has been made to overlay a word in the control store or an entry in the map tables. These messages are suppressed by giving the options -m and -i on the command line.

Global name already defined
: An attempt has been made to define the global symbol name more than once.

Illegal default entry point n
: The default entry number n is not a multiple of the map table page size.

Default entry point n defined twice
: There is more than one definition of the specified default entry point.

More than one control store page loaded
: There is code loaded into more than one control store page. Great care is needed in such cases because branches other than via the map tables may go to the wrong page.

Attempt to change control store width to n
: A T.MICWIDTH record has been encountered which attempts to change the width of the control store.

Attempt to change control store length to n
: A T.MICLEN record has been encountered which attempts to change the length of the control store.

Attempt to change map table width to n
: A T.ENTWIDTH record has been encountered which attempts to change the width of the map tables.

Attempt to change map table length to n
: A T.ENTLEN record has been encountered which attempts to change the length of the map tables.

Attempt to change control store page size to n
: A T.MICPAGE record has been encountered which attempts to change the page size in the control store.

Attempt to change map table page size to n
: A T.ENTPAGE record has been encountered which attempts to change the page size in the map tables.

Attempt to redefine parity field
: A parity definition record (T.MICEVENPARITY, T.MICODDPARITY, T.ENTEVENPARITY, or T.ENTODDPARITY) has been encountered in which the new data differs from the old.

APPENDIX D

Mb Microlibrarian Users Manual

Mb combines and maintains groups of relocatable binary files produced by the microassembler ms in a single library file. Its main uses are to create or update library files used by the linker program ml and to list their contents.

D.1 Librarian operation

The general form of invoking mb at OTS command level is

mb [-o outfile] libfile options files

All files are of type .mrl. libfile is the library file. files are the names of one or more constituent segments of the library. These will be segments written in microassembly language. outfile is an optional output file. The various options are described below.

It is important to understand what is meant by a module. A module is a unit of relocatable code contained in a single file and assembled to a file of type .mrl. A module has a name (used only by mb) which is derived from the file name or from a NAME statement in the source code. Depending upon context, a module name may be specified either as a string of up to 10 letters or digits, or as a full file specification. The file type is assumed to be .mrl; if a file type is specified it is ignored. Where the file name specifies a module within the library, a directory name is meaningless and is ignored.

The most commonly used option, -u, updates one or more modules in the library. mb searches libfile.mrl for an occurrence of file, replacing the library version with a fresh copy from file.mrl. If file is not in the library a new copy is inserted at the end.

Normally no outfile is specified and the new version of the library is named libfile.mrl, the old version being renamed libfile.bak~. If the -o option is given, the new version of the library is called outfile.mrl, leaving the old version unchanged.

mb is very conservative. If problems are detected, e.g. a file cannot be found, the library file is not altered. The options are

-u Update the named modules in the library. Any occurrences of the named modules in the library are deleted and replaced with new versions. If the named modules are not already in the library they are inserted at its end. If the library file does not exist, it is created. If no names are given, all the modules in the library are updated. -u is the default if option is omitted but one or more module files are specified.

If the optional form -ub posfile is used, the named files are inserted before the library module posfile. If the form -ua posfile is used, they are inserted after posfile. In both cases, pre-

existing versions of the named module files are deleted, wherever they are in the library (but posfile is not touched).

-d Delete the named files from the library.

-x Extract the named files from the library. If no names are given, all the module files in the library are extracted. In neither case is the library altered.

-t Display a listing of the contents of the library on the user's terminal. The listing consists of the name of each module in the library, accompanied by a list of names within it which have been declared global. -t is the default when no option and no segfiles are specified.

If the optional form -tv is used, the listing is verbose; the name of each constituent file is displayed together with a list of global names as with option -t. This is followed by the size of the relocatable segment, the size of the absolute segment (if present), the number of defined entry points, and a list of external names that are referenced but not defined within this module.

The form -tf is similar to -tv, directing its output to a listing file libfile.prn.

-s Scan libfile for unresolved forward references. In order that the linker need make only one pass through a library, no module should refer to an external name that is defined in an earlier module of the library.

When developing a large program consisting of many modules, mb can also be used to maintain a single .mrl file containing some or all of the modules that comprise the program. Strictly speaking, this is a file of concatenated object files rather than a library, since the file will be presented to the linker in its entirety instead of as a library from which to extract selected modules.

D.2 Error messages

The diagnostics produced by mb are intended to be self explanatory.

D.3 Examples

mb mylib -u test1 test2

Update the modules test1 and test2 in the library file mylib.mrl from files test1.mrl and test2.mrl. If test1 or test2 is not present in the library, insert it at the end. (The command mb mylib test1 test2 is equivalent.)

mb mylib -ub test2 newseg

Insert the file newseg.mrl into the library file mylib.mrl before module test2. Delete any pre-existing occurrences of newseg.

mb mylib -u

Update all modules in mylib. If any module file cannot be found in the current directory, a warning is output and mylib is not changed.

mb blib -t
Output a listing of the contents of blib.mrl to the user's terminal. (The command mb blib is equivalent.)

APPENDIX E

Standard Instruction Set Listings

This appendix provides a source listing of selected modules from the standard instruction set. It provides real examples of significant sections of microcode. When studied in conjunction with Chapter 10 it illustrates the conventions which must be followed when writing new code to be added to the standard system.

These examples are provided for illustration only and may differ in detail from the standard system as currently distributed.

```
/*
        register.m  --  Register definitions

        Copyright (c) 1983 by High Level Hardware Limited
*/

ir0     =       R5
ir1     =       R6
pr      =       R8
psw     =       R9
vfp     =       R10
cb      =       R11
vsp     =       R12
fp      =       R13
sp      =       R14
pc      =       R15
```

```
/*
        rinit  --  DTROM bootstrap microcode

        Copyright (c) 1983 by High Level Hardware Limited
*/

*I vahdr

// No registers in use at this point.

        physadd         =       R0
        logadd          =       R1
        protmask        =       R2
        curradd         =       R3
        cachebank       =       R4
        temp            =       R5
        bytereg         =       R6

// System mode instruction set and cache bank.

        syscache        =       0
        sysinst         =       15

// Interesting constants.

        cachesize       =       512
        maxbank         =       15
        privreg         =       0
        slotsize        =       2       // SHL2
        maxslots        =       511
        maxlogadd       =       8H      // SHL2
        biocid          =       1

// Standard startup, initialize sequencer.

        CJP     NLC, @1
        JZ      F

@1:

// Write to the whole of the cache to remove potential parity errors.

        CONT    DZ D=BR, maxbank  OR  RAMF B=cachebank
@2:
                PUSH    cachesize/2 - 1  ZA A=cachebank  OR  HLDCA
                CONT    ZA  AND  LDCA
                        RFCT    ZA  AND  CWR  INCCA
                        CONT    ZA  AND  CWR  INCCA
                PUSH    cachesize/2 - 1  ZA A=cachebank  OR  HLDCA
                CONT    ZA  AND  ALDCA
                        RFCT    ZA  AND  CWR  INCCA
                        CONT    ZA  AND  CWR  INCCA
                CJP     NZ, @2  ZA A=cachebank  OR
```

```
                CONT    ZB  SUBR  RAMF B=cachebank

// Select privileged registers, location 0.

        CONT    DZ D=BR, privreg  OR  HLDCA
        CONT    ZA  AND  ALDCA

// Initialize all memory management regions.  4K pages, no region fault.

        CONT    DZ D=BR, set.4k.pages | nregf | region0 SHL3  WRMM
        CONT    DZ D=BR, set.4k.pages | nregf | region1 SHL3  WRMM
        CONT    DZ D=BR, set.4k.pages | nregf | region2 SHL3  WRMM
        CONT    DZ D=BR, set.4k.pages | nregf | region3 SHL3  WRMM

/*

Write  to  the whole of the translation buffer to remove potential parity
errors.  Set bounds bit on all locations.

*/

        PUSH    tablesize/4 - 1  ZB  AND  RAMF B=logadd
        CONT    DZ D=BR, b SHL3  OR  RAMF B=protmask

                CJS     writetb
                CONT    ZA  AND  RAMF B=physadd
                RFCT    DA D=BR, size.4k.page SHL1 A=logadd  \
                                 ADD  RAMF B=logadd
                CONT

/*

Allow access to entry 0 so that physical addressing can be used.  We must
find  some  physical  memory  out  there  so this entry will be rewritten
anyway.

*/

        CJS     writetb  ZB  AND  RAMF  B=logadd
        CONT    DZ D=BR, a | m | nr | v | nb SHL3  OR  RAMF B=protmask

retry:

/*

Inspect  all  slots  and  determine contents.  Store result in privileged
register with the same number as the slot.  If we find memory,  clear  it
and initialize entries in the translation buffer.

*/

        PUSH    maxslots  ZB  AND  RAMF B=physadd
        CONT    ZB  AND  RAMF B=logadd
```

```
/*

Write zero and read it back.  Memory reads as zero, empty slot as -1  and
I/O subsystem something else depending on type.

*/

                CONT    physadd  LBR  ADDR
                CONT    ZA  AND  LBR  WR
                CONT    LOCK
                CONT    physadd  LBR  ADDR
                CONT    LOCK
                CONT    RD
                CONT

                CJS     Z, clearmem  DZ D=BUS  OR  CWR
                CONT
                RFCT    DA D=BR, slotsize SHL2 A=physadd  \
                                ADD  RAMF B=physadd  INCCA

// Clear any parity errors.

                CONT    CLRPERR

// Look for booting IOC.

        PUSH    cachesize - 1  ZB  AND  RAMF B=physadd  ALDCA
        CONT    DZ D=BR, biocid  OR  RAMF B=temp
                TWB     Z, retry  DA D=CSH MASK A=temp  EXOR  INCCA
                CONT    DA D=BR, slotsize SHL2 A=physadd  \
                                ADD  RAMF B=physadd

// Have found IOC send byte to indicate ready

        CONT    DA D=BR, slotsize SHL2 A=physadd  SUBR CIN \
                RAMF B=physadd  INCCA
        CJS     ioc.out
        CONT    DZ D=BR, biocid  OR  RAMF B=bytereg

// Now copy data from ioc to DT.

dataloop:
                CJS     ioc.in
                CONT
                CJS     dt.out
                CONT
                CJP     dataloop
                CONT
//----------

clearmem:

/*
```

```
Clear slotsize full of memory.  If  there  is  room  in  the  translation
buffer, make appropriate entries.

*/

        CONT    ZA A=physadd  OR  RAMF B=curradd
        CONT    DA D=BR, slotsize SHL2 A=physadd  ADD  QREG
@1:
                CONT    ZB  ADD CIN  RAMA A=physadd B=physadd  LBR  ADDR
                CONT    ZB  ADD CIN  RAMF B=physadd  D=BR,0  LBR  WR
                CONT    RWR
                CJP     NZ, @1  AQ A=physadd  EXOR  LOCK
                CONT

// Return if translation buffer is full.  Restore physadd.

        CRTN    BW  DA D=BR, maxlogadd SHL2 A=logadd  SUB
        CONT    ZA A=curradd  OR  RAMF B=physadd

// Otherwise enter into translation buffer.  Protmask is already set up.

@2:
                CJS     writetb
                CONT
                CONT    DA D=BR, size.4k.page SHL1 A=physadd  \
                                ADD  RAMF  B=physadd
                CJP     NZ, @2 AQ A=physadd  EXOR
                CONT    DA D=BR, size.4k.page SHL1 A=logadd  \
                                ADD  RAMF  B=logadd

// Return, regenerating physadd.

        CRTN    ZA A=curradd  OR  RAMF B=physadd
        CONT
//---------

writetb:

/*

Enter  with  physical  address in physadd, logical address in logadd, and
protection mask in protmask.  Write to all four regions.

*/

        CONT    DA D=BR, region0 SHL3 A=logadd  OR  LVAR
        CONT    AB A=physadd B=protmask  OR  TBWR
        CONT    DA D=BR, region1 SHL3 A=logadd  OR  LVAR
        CONT    AB A=physadd B=protmask  OR  TBWR
        CONT    DA D=BR, region2 SHL3 A=logadd  OR  LVAR
        CONT    AB A=physadd B=protmask  OR  TBWR
        CRTN    DA D=BR, region3 SHL3 A=logadd  OR  LVAR
        CONT    AB A=physadd B=protmask  OR  TBWR
```

```
//---------

dt.in:

// Input byte returned in bytereg.

// Test ready bit (bit 15 in CAIR).

in1:    CJP     NS, in1  DZ D=CAIR SHL2  OR
        CONT

// Allow time for port to turn on.

        CONT    ININT   S7
        CONT    ININT   S7
        CONT    DZ D=CAIR  ZZZD  OR  RAMF B=bytereg      ININT  S7

// Allow time for READY to drop.

        CONT    S7
        CONT    S7
        CRTN    S7
        CONT    S7
//----------

dt.out:

// Least significant byte of bytereg output.

        CONT    ZB B=bytereg OR LDIR

// Test ready bit (bit 14 in CAIR).

out1:   CONT    DZ D=CAIR SHL2  OR  RAMU B=bytereg
        CJP     NS, out1  bytereg
        CONT

// Allow time for strobe.

        CONT    OUTINT  S7
        CONT    OUTINT  S7
        CONT    OUTINT  S7

// Allow time for READY to drop.

        CONT    S7
        CONT    S7
        CONT    S7
        CRTN    S7
        CONT    S7
//----------

ioc.in:
```

```
// Input byte returned in bytereg.

// Test ready bit (bit 7 in status register).

@1:
                CONT    ZA A=physadd  OR  LBR  ADDR
                CONT    RRD
                CONT
                CJP     @1, NS  DZ D=BUS SHL3  OR
                CONT
        CONT    ADDR
        CONT    RD
        CRTN
        CONT    DZ D=BUS MASK  OR  RAMF B=bytereg
//---------

ioc.out:

// Output byte in bytereg.

// Test ready bit (bit 0 in status register).

@1:
                CONT    ZA A=physadd  OR  LBR  ADDR
                CONT    RRD
                CONT
                CJP     @1, EV  DZ D=BUS  OR
                CONT
        CRTN    ADDR
        CONT    ZA A=bytereg  OR  LBR  WR
//---------
```

```
/*
        fetch  --  Instruction fetch

        Copyright (c) 1982 by High Level Hardware Limited
*/

*I      register.m
*I      opcodes.m
*I      trapcodes.m

EXTERNAL        trap, rcache, decpc

ENTRY   umode + refill
ENTRY   kmode + refill

/*

Come here after a control transfer, i.e. after a FETCH.  Test to  see  if
an  interrupt  is  pending.   If it is then test to see if interrupts are
enabled.  If so then  restore  pc  and  jump  to  trap,  passing  in  the
interrupt  trap  code.   If  interrupts  are  disabled, or if there is no
interrupt, join the main line code.

*/
        CJP     NINT @2  ZB  ADD CIN  RAMA A=pc B=pc  LVAR
        CJP     EV @3   ZA A=psw  OR  D=TB  LBR  ADDR
        CJP     PE, except  LOCK

/*

If  we  reach  here  an  interrupt is pending and interrupts are enabled.
Unless a parity error has been flagged we must service  it.   The  memory
access  may be aborted but there must be a LOCK to prevent a short memory
cycle.  Note that ir0  and  ir1  are  undefined  after  the  FETCH  which
transferred  control  here  but  that  on return from interrupt the first
instruction to be executed will be taken from the least significant  byte
of  ir0.   Hence we store refill in ir0 so that the code is reloaded into
ir0 and ir1.  pc must also be decremented so that control continues  from
this point.

*/
        CONT    DZ D=BR,refill  OR  RAMF B=ir0  LOCK
        JUMP    trap    ZB  SUBR  RAMF B=pc
        CONT    DZ D=BR, interrupt+256  OR  QREG
//----------

ENTRY   umode + fetch
ENTRY   kmode + fetch

/*

Come here when a word of code is  exhausted.   If  pc  is  odd  then  ir1
contains  the  next  word  of (pre-fetched) code which can be transferred
```

```
into ir0.  If even we must fetch two words from memory.  In  either  case
increment pc by 1.

*/
        CONT    DZ SHR1  OR  RAMA A=ir1 B=ir0  LDIR
        CJV     OD  ZB  ADD CIN  RAMA A=pc B=pc  LVAR
        CONT    D=TB  LBR  ADDR                  // NB ADDR after CJV!

@2:

/*

We arrive here either by falling through, in which case pc must  be  even
and 2 words are required or else after a FETCH, in which case pc might be
odd.  Since the second word is free, and  the  content  of  ir1  will  be
ignored  if  pc  is  odd we always fetch two words, the first in ir0, the
second in ir1.  If any type of parity error has been  flagged  branch  to
except,  which  will  determine the true cause of the fault.  If the code
fetch produces a RFLT, call rcache and load refill into IR so  that  code
is  fetched  if  the problem can be patched up.  Remember to decrement pc
again.

*/

        CJP     PE, except  LOCK
@3:
        CONT    DZ D=BR, refill  OR  RAMF B=R0  RD
        CJP     RFLT, @4  RRD
        CONT    DZ D=BUS  OR  RAMF B=ir0  LDIR
        CJV     DZ D=BUS  OR  RAMF B=ir1

// Remove a byte from ir0, the same as any normal instruction.

        CONT    DZ  SHR1  OR  RAMA A=ir0 B=ir0
//----------

@4:
        CJS     rcache  ZA A=R0  OR  LDIR
        LDCT    decpc
        CONT    RRD
        CONT    DZ D=BUS  OR  RAMF B=ir0  LDIR
        CJV     DZ D=BUS  OR  RAMF B=ir1

// Remove a byte from ir0, the same as any normal instruction.

        CONT    DZ  SHR1  OR  RAMA A=ir0 B=ir0
//----------

except:

/*

We  arrive  here if any type of parity error has been detected.  Test for
```

```
each condition in turn and pass an appropriate trap code.  Restore pc for
possible use later.

*/

// Main memory parity error.

        CJP     MP, @1  ZB  SUBR  RAMF B=pc
        CONT    DZ D=BR, mparitytrap  OR  QREG

// Translation buffer parity error.

        CJP     TBP, @1
        CONT    DZ D=BR, tbparitytrap  OR  QREG
//----------

// Cache memory parity error if we reach here.

        CONT    DZ D=BR, cparitytrap  OR  QREG


@1:

/*

Clear  down  parity  errors.  Note  that  if  more  than  one  occurs
simultaneously only the first to  be  checked  will  be  detected.  This
presumably happens only rarely!

*/

        JUMP    trap  CLRPERR
        CONT
//----------
```

```
/*
        lp  --  Load parameter

        Copyright (c) 1984 by High Level Hardware Limited

*/

*I      register.m
*I      opcodes.m

EXTERNAL        ftod, trap


/*

Special cases are provided for lp_w 3 to lp_w 10  so  that  more  compact
code  may  be generated for these commonly used instructions.  All are of
the same form.  The first instruction branches to a  common  end  routine
and  also  reloads the instruction register and ir0.  The second adds the
constant offset to the frame pointer and loads the cache address with the
result  so that the value may be stored at the cache location holding the
parameter.

*/

ENTRY   umode + lp3
ENTRY   kmode + lp3

        JUMP    @1      DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
        CONT    DA D=BR,3 A=fp  ADD  LDCA
//----------

ENTRY   umode + lp4
ENTRY   kmode + lp4

        JUMP    @1      DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
        CONT    DA D=BR,4 A=fp  ADD  LDCA
//----------

ENTRY   umode + lp5
ENTRY   kmode + lp5

        JUMP    @1      DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
        CONT    DA D=BR,5 A=fp  ADD  LDCA
//----------

ENTRY   umode + lp6
ENTRY   kmode + lp6

        JUMP    @1      DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
        CONT    DA D=BR,6 A=fp  ADD  LDCA
//----------
```

```
ENTRY   umode + lp7
ENTRY   kmode + lp7

        JUMP    @1      DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
        CONT    DA D=BR,7 A=fp  ADD  LDCA
//----------

ENTRY   umode + lp8
ENTRY   kmode + lp8

        JUMP    @1      DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
        CONT    DA D=BR,8 A=fp  ADD  LDCA
//----------

ENTRY   umode + lp9
ENTRY   kmode + lp9

        JUMP    @1      DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
        CONT    DA D=BR,9 A=fp  ADD  LDCA
//----------

ENTRY   umode + lp10
ENTRY   kmode + lp10

        JUMP    @1      DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
        CONT    DA D=BR,10 A=fp  ADD  LDCA
//----------

/*

The  general  case is very similar to the previous ones.  The argument is
extracted from the code stream and saved in R0 by  a  PLDIR,  which  also
reloads the instruction register.  The second instruction removes the two
surplus bytes from ir0 (the lp_w opcode and the parameter number).

*/

ENTRY   umode + lp_w
ENTRY   kmode + lp_w

        CONT    DZ MASK  OR  RAMA A=ir0 B=R0  PLDIR
        JUMP    @1      DZ SHR2  OR  RAMA A=ir0 B=ir0
        CONT    AB  A=fp B=R0  ADD  LDCA
//----------

@1:

/*

The common ending writes the parameter onto the stack, incrementing the
stack pointer and setting the cache address appropriately.

*/
```

```
        CJV     ZB  ADD CIN  RAMF B=sp  LDCA
        CONT    CSH  CWR
//----------
```

```
/*

lp_c, lp_h and lp_s are the same as lp_w with the exception that the
parameter must be masked (and sign extended for lp_s). The value must be
stored temporarily as CWR operates on the D bus, and this is needed for
bringing in the data. The instruction to write the stored value to the
cache is shared between several high level instructions

*/

ENTRY   umode + lp_c
ENTRY   kmode + lp_c

        CONT    DZ MASK  OR  RAMA A=ir0 B=R0  PLDIR
        CONT    DZ SHR2  OR  RAMA A=ir0 B=ir0
        CONT    AB  A=fp B=R0  ADD  LDCA
        JUMP    @3      ZB  ADD CIN  RAMF B=sp  LDCA
        CJV     DZ CSH MASK  OR  RAMF B=R0
//----------


ENTRY   umode + lp_s
ENTRY   kmode + lp_s

/*

The sign extension introduces a little complexity here as the sign bit is
in the middle of the word in bit 15. Consequently the low half-word is
read into the high half, where its sign can be tested on the CJP
instruction. In both cases the following instruction returns the short
to the low half-word; if the value is non-negative the CJV transfers
control and the data is written to the cache on the final instruction.
When the value is negative the code at @2 is reached. This OR's in the
sign extension and writes the extended value to the cache.

*/

        CONT    DZ MASK  OR  RAMA A=ir0 B=R0  PLDIR
        CONT    DZ SHR2  OR  RAMA A=ir0 B=ir0
        CONT    AB  A=fp B=R0  ADD  LDCA
        LDCT    @3      ZB  ADD CIN  RAMF B=sp  LDCA
        JRP     S, @2   DZ CSH CDZZ  OR  RAMF B=R0
        CJV     NLC     DZ ZZAB  OR  RAMA A=R0 B=R0
@2:
        CJV     DA D=BR,-1 ABZZ A=R0  OR  RAMF B=R0
@3:
        CONT    R0  CWR
//----------
```

```
ENTRY   umode + lp_h
ENTRY   kmode + lp_h

/*

lp_h is straightforward.  The 16 high order bits are simply masked off.

*/

        CONT    DZ MASK  OR  RAMA A=ir0 B=R0  PLDIR
        CONT    DZ SHR2  OR  RAMA A=ir0 B=ir0
        CONT    AB  A=fp B=R0  ADD  LDCA
        JUMP    @3       ZB  ADD CIN  RAMF B=sp  LDCA
        CJV     DZ D=CSH ZZCD  OR  RAMF B=R0
//----------


ENTRY   umode + lp_d
ENTRY   kmode + lp_d

/*

As  in  lp_w, lp_d begins by extracting the argument from the code stream
with a  PLDIR  and  discarding  surplus  bytes  from  ir0.  The  third
instruction  addresses  the most significant word, ready for reading into
R0 on the fifth which also  sets  the  cache  address  to  point  to  the
location  where  the  least  significant word will be loaded.  The fourth
instruction increments the stack pointer in preparation for a double word
push  which  is  completed on the last line.  The CJV increments sp again
and writes the LSW, incrementing the cache address  for  the  MSW  to  be
written on the following line.

*/

        CONT    DZ MASK  OR  RAMA A=ir0 B=R0  PLDIR
        CONT    DZ SHR2  OR  RAMA A=ir0 B=ir0
        CONT    AB A=fp B=R0  ADD CIN  LDCA
        CONT    ZB  ADD CIN  RAMF B=sp  DECCA
        JUMP    @3       DZ CSH  OR  RAMA B=R0  A=sp LDCA
        CJV     ZB  ADD CIN  RAMF B=sp  CSH CWR  INCCA
//----------


ENTRY   umode + lp_f
ENTRY   kmode + lp_f

/*

As before we obtain the parameter from the code stream and reload IR with
the PLDIR function and add the parameter to fp and set the cache address.
We  delay  shifting  ir0  as it is not yet necessary and by performing it
later we can save a microinstruction of code (though not  in  execution).
When  resetting the cache address from the stack pointer call an EXTERNAL
```

routine to convert the float read into R0 on the next cycle to a double in R0 R1. After returning we shift ir0 and CJV to the next operation, writing out the less significant word from R1 to the cache and incrementing sp and CA again. Finally we write the more significant word to the stack, using the shared instruction at @3.

*/

```
        CONT    DZ MASK  OR  RAMA A=ir0 B=R0  PLDIR
        CONT    AB  A=fp B=R0  ADD  LDCA
        CJS     ftod            ZB  ADD CIN  RAMF B=sp  LDCA
        CONT    DZ D=CSH  OR  RAMF B=R0

        CJP     NZ, invalid     AB A=R2 B=psw  AND
        CJP     NLC, @3         ZB  ADD CIN  RAMA A=R1 B=sp  CWR  INCCA
        CJV     DZ SHR2  OR  RAMA A=ir0 B=ir0
//----------

invalid:

        CONT    DZ SHR1  OR  RAMA A=ir0 B=ir0
        JUMP    trap    R0  CWR
        CONT    DZ D=BR, lp_f  OR  LDIR
//----------


ENTRY   umode + llp
ENTRY   kmode + llp

/*

llp is essentially the first four instructions of lp_w. However, the value written to the top of the stack is the address obtained by adding the argument offset to the frame pointer, rather than the value stored at that location.

*/

        CONT    DZ MASK  OR  RAMA A=ir0 B=R0  PLDIR
        CONT    DZ SHR2  OR  RAMA A=ir0 B=ir0  INCCA
        CJV     AB A=fp B=R0  ADD  RAMF
        CONT    ZB  ADD CIN  RAMA A=R0 B=sp  CWR
//----------
```

```
/*
        ln  --  Load number

        Copyright (c) 1983 by High Level Hardware Limited

*/

*I      register.m
*I      opcodes.m

EXTERNAL        rcache, decpc


/*

The operations to load -1, 0, 1, 2, a byte argument and the negative of a
byte  argument  are all very similar.  The first microinstruction reloads
the  instruction  register  and  jumps  to  a  common  instruction  which
increments  the  stack  pointer and writes the value to the cache.  In the
case of the load constant operations this instruction also removes the
opcode from the code stream.  In the case of lnb and lnmb the argument is
removed from the code stream with a PLDIR and the surplus  bytes  removed
from ir0 on the following instruction.

*/

ENTRY   umode + lnm1
ENTRY   kmode + lnm1

        JUMP    @1      DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
        CJV     DZ D=BR,-1  OR  RAMF B=R0  INCCA
//----------

ENTRY   umode + ln0
ENTRY   kmode + ln0

        JUMP    @1      DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
        CJV     DZ D=BR,0  OR  RAMF B=R0  INCCA
//----------

ENTRY   umode + ln1
ENTRY   kmode + ln1

        JUMP    @1      DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
        CJV     DZ D=BR,1  OR  RAMF B=R0  INCCA
//----------

ENTRY   umode + ln2
ENTRY   kmode + ln2

        JUMP    @1      DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
        CJV     DZ D=BR,2  OR  RAMF B=R0  INCCA
//----------
```

```
ENTRY   umode + lnb
ENTRY   kmode + lnb

        JUMP    @1      DZ MASK  OR  RAMA A=ir0 B=R0  PLDIR
        CJV     DZ SHR2  OR  RAMA A=ir0 B=ir0  INCCA
//----------

ENTRY   umode + lnmb
ENTRY   kmode + lnmb

        JUMP    @1      DZ MASK  SUBR CIN  RAMA A=ir0 B=R0  PLDIR
        CJV     DZ SHR2  OR  RAMA A=ir0 B=ir0  INCCA
@1:
        CONT    ZB  ADD CIN  RAMA A=R0 B=sp  CWR
//----------

ENTRY   umode + lnw
ENTRY   kmode + lnw

/*
```

The word argument for this operation is also provided from the code stream but a complication not present in lnb and lnmb is that the argument may not have been fetched. If the program counter is odd the argument has been fetched and currently resides in irl. Note, however, that the program counter must be incremented to step over the argument.

If the argument is not in irl we must read it from memory.
Assuming there is no memory fault, the first word read from memory is written to the cache and the second written to irl, thereby pre-fetching some more code. If a memory read fault is detected the operation is aborted by branching to @3.

```
*/

        CJP     OD, @2  ZB  ADD CIN  RAMA A=pc B=pc  INCCA  LVAR
        CONT    ZA A=ir0  OR  LDIR  D=TB  LBR  ADDR
        CJS     RFLT, rcache  LOCK
        LDCT    @3  RD
        CONT    DZ SHR1  OR  RAMA A=ir0 B=ir0  RRD
        CJV     ZB  ADD CIN  RAMF B=sp  D=BUS  CWR
        CONT    DZ D=BUS  OR  RAMF B=irl
@2:
        CJV     DZ SHR1  OR  RAMA A=ir0 B=ir0
        CONT    ZB  ADD CIN  RAMA A=irl B=sp  CWR
//----------

@3:

/*
```

By this time the instruction register has been loaded with the next instruction. The program counter has been incremented; the last two

```
instructions correct these registers.

*/

        CJP     decpc  DECCA
        CONT    DZ D=BR,lnw  OR  LDIR
//----------
```

```
/*
        rv  --  Load right value

        Copyright (c) 1984 by High Level Hardware Limited

*/

*I      register.m
*I      opcodes.m

        b_mask  =       0CH     // Byte address bits (SHL3).
        c_mask  =       511     // Size of cache bank.

EXTERNAL        ftod, trap, rcache, noop

/*

The  rv_x  operations take an address off the TOS and reload the TOS with
the value found at  that  address.  As  the  address  is  calculated  at
run-time the value may be cached; this must be checked for.

*/

ENTRY   umode + rv_c
ENTRY   kmode + rv_c

/*

*/

        CJP     OB, @c1 DZ D=CSH  OR  RAMF B=R0  LVAR  LDCA
        CONT    OS      DA D=BR,b_mask SHL3  NOTRS  RAMA A=R0 B=R1
        CJP     LC, @c2 AB A=cb B=R1  SUBR CIN  QREG  D=TB  LBR  ADDR
        CONT    Z       DQ D=BR,c_mask  NOTRS  LOCK
        CJP     LC, @c4 DZ D=CAIR  MASK  OR  RAMF B=R2  RD
        JUMP    @c8     DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
@c23:
        CJV     NRFLT   DZ D=BUS ZZZD  OR  RAMA A=sp B=R1  LDCA
//----------

@c1:

        CJP     LC, @c3 AB A=cb B=R1  SUBR CIN  QREG  D=TB  LBR  ADDR
        CONT    Z       DQ D=BR,c_mask  NOTRS  LOCK
        CJP     LC, @c5 DZ D=CAIR  MASK  OR  RAMF B=R2  RD
        JUMP    @c8     DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
@c24:
        CJV     NRFLT   DZ D=BUS ZZZC  OR  RAMA A=sp B=R1  LDCA
//----------

@c2:
        CJP     LC, @c6  DZ D=CAIR  MASK  OR  RAMF B=R2  RD
        JUMP    @c8     DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
```

```
@c21:
        CJV     NRFLT   DZ D=BUS ZZZB  OR  RAMA A=sp B=R1  LDCA
//----------

@c3:
        CJP     LC, @c7  DZ D=CAIR  MASK  OR  RAMF B=R2  RD
        JUMP    @c8      DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
@c22:
        CJV     NRFLT   DZ D=BUS ZZZA  OR  RAMA A=sp B=R1  LDCA
//----------

@c4:
        CJV     DZ D=CSH ZZZD  OR  RAMA A=sp B=R1  LDCA
@c5:
        CJV     DZ D=CSH ZZZC  OR  RAMA A=sp B=R1  LDCA
@c6:
        CJV     DZ D=CSH ZZZB  OR  RAMA A=sp B=R1  LDCA
@c7:
        CJV     DZ D=CSH ZZZA  OR  RAMA A=sp B=R1  LDCA
//----------

@c8:
        CJS     NLC, rcache  ZA A=R1  OR  CWR
        LDCT    @fixup  ZA A=R0  OR  CWR

        CJP     OB, @c20  ZA A=R0  OR
        LDCT    @c23
        JRP     OS, @c21  ZA A=R0  OR
        JUMP    @c8
@c20:
        CJP     OS, @c22  ZA A=R0  OR
        JUMP    @c8
        CJV     DZ D=BUS ZZZC  OR  RAMA A=sp B=R1  LDCA
//----------


ENTRY   umode + rv_s
ENTRY   kmode + rv_s

/*

*/

        LDCT    @s5      DZ D=CSH  OR  RAMF B=R0  LVAR  LDCA
        CONT    OS       DA D=BR,b_mask SHL3  NOTRS  RAMA A=R0 B=R1
        CJP     LC, @s1 AB A=cb B=R1  SUBR CIN  QREG  D=TB  LBR  ADDR
        CONT    Z        DQ D=BR,c_mask  NOTRS  LOCK

        CJP     LC, @s2  DZ D=CAIR MASK  OR  RAMF B=R2  RD
        CONT    DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
@s21:
        JRP     S, @s3  DZ D=BUS CDZZ  OR  RAMA A=sp B=R1  LDCA
        CJP     @s20  DZ ZZAB  OR  RAMA A=R1 B=R1
```

```
//----------

@s2:
        JRP     S, @s3  DZ D=CSH CDZZ  OR  RAMA A=sp B=R1  LDCA
        CJP     @s20  DZ ZZAB  OR  RAMA A=R1 B=R1
//----------

@s1:
        CJP     LC, @s4  DZ D=CAIR MASK  OR  RAMF B=R2  RD
        CONT    DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
@s22:
        JRP     S, @s3  DZ D=BUS ABZZ  OR  RAMA A=sp B=R1  LDCA
        CJP     @s20  DZ ZZAB  OR  RAMA A=R1 B=R1
//----------

@s4:
        CJP     S, @s3  DZ D=CSH ABZZ  OR  RAMA A=sp B=R1  LDCA
        CJP     @s20  DZ ZZAB  OR  RAMA A=R1 B=R1
//----------

@s5:
        CJV     NRFLT
@s3:
        CJV     NRFLT  DA D=BR,-1 ABZZ A=R1  OR  RAMF B=R1
@s20:
        CJS     NLC, rcache  ZA A=R1  OR  CWR
        LDCT    @fixup  ZA A=R0  OR  CWR
        LDCT    @s21
        JRP     OS, @s22  ZA A=R0  OR
        LDCT    @s5
//----------

ENTRY   umode + rv_h
ENTRY   kmode + rv_h

/*

rv_h is essentially rv_s without the sign extension code.

*/

        CONT    DZ D=CSH  OR  RAMF B=R0  LVAR  LDCA
        CONT    OS      DA D=BR,b_mask SHL3  NOTRS  RAMA A=R0 B=R1
        CJP     LC, @h1 AB A=cb B=R1  SUBR CIN  QREG  D=TB  LBR  ADDR
        CONT    Z       DQ D=BR,c_mask  NOTRS  LOCK

        CJP     LC, @h2  DZ D=CAIR MASK  OR  RAMF B=R2  RD
        JUMP    @h4     DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
@h21:
        CJV     NRFLT   DZ D=BUS ZZCD  OR  RAMA A=sp B=R1  LDCA
//----------

@h1:
```

```
        CJP     LC, @h3  DZ D=CAIR MASK  OR  RAMF B=R2  RD
        JUMP    @h4      DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
@h22:
        CJV     NRFLT    DZ D=BUS ZZAB  OR  RAMA A=sp B=R1  LDCA
//----------

@h2:
        CJV     DZ D=CSH ZZCD  OR  RAMA A=sp B=R1  LDCA
@h3:
        CJV     DZ D=CSH ZZAB  OR  RAMA A=sp B=R1  LDCA
//----------

@h4:
        CJS     NLC, rcache  ZA A=R1  OR  CWR
        LDCT    @fixup  ZA A=R0  OR  CWR
        LDCT    @h21
        JRP     OS, @h22  ZA A=R0  OR
        CJP     @h4
//----------


ENTRY   umode + rv_w
ENTRY   kmode + rv_w

/*

For rv_w it is more efficient to test for the address being below the
cache base or above the TOS separately, branching out to read from memory
if necessary.  If the value must come from the cache, CA is loaded with
the address in R0, then with sp to re-address the TOS.  Because of the
pipeline delay it is possible for the final instruction to read from the
first address and write to the second.

*/

        CONT    DZ D=CSH  OR  RAMF B=R0  LVAR
        CJP     NBW, @w1        AB A=cb B=R0  SUB  D=TB  LBR  ADDR
        CJP     NBW, @w2        AB A=sp B=R0  SUBR  LOCK
        CONT    DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR  RD
        CONT    R0  LDCA
        CJV     sp  LDCA
        CONT    CSH  CWR
//----------

@w1:
        CONT    DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR  RD
@w2:
        CJV     NRFLT  DZ D=BR, rv_w  OR  RAMF B=R2
        CJS     NLC, rcache  D=BUS  CWR
        LDCT    @fixup  ZA A=R0  OR  CWR
        CJV
        CONT    D=BUS  CWR
//----------
```

```
ENTRY   umode + rv_f
ENTRY   kmode + rv_f

/*

As for rv_w to get the float, then use ftod to convert to a double,
finally writing it to the stack and incrementing sp.

*/

        LDCT    rcache  DZ D=CSH  OR  RAMF B=R0  LVAR
        CJP     NBW, @f1        AB A=cb B=R0  SUB  D=TB  LBR  ADDR
        CJP     NBW, @f2        AB A=sp B=R0  SUBR  LOCK
        CONT    DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR  RD
        CONT    R0  LDCA
        CJP     @f4     sp  LDCA
        CJS     ftod            DZ D=CSH  OR  RAMF B=R0
//----------

@f1:
        CONT    DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR  RD
@f2:
        CJP     NRFLT, @f4
        CJS     LC, ftod  DZ D=BUS  OR  RAMF B=R0

        JSRP    F  DZ D=BR, rv_f  OR  RAMF B=R2
        LDCT    @fixup
        CJS     ftod
        CONT    DZ D=BUS  OR  RAMF B=R0
@f4:
        CONT
        CONT    ZB  ADD CIN RAMA A=R1 B=sp  CWR  INCCA
        CJV     Z       AB A=R2 B=psw  AND
        CONT    R0  CWR
        CONT    DZ  SHL1  OR  RAMA A=ir0 B=ir0
        JUMP    trap    DA D=CAIR MASK A=ir0  OR  RAMF B=ir0
        CONT    R2  LDIR
//----------

@fixup:
        CONT    DZ  SHL1  OR  RAMA A=ir0 B=ir0
        JUMP    noop    DA D=CAIR MASK A=ir0  OR  RAMF B=ir0
        CONT    R2  LDIR
//----------


ENTRY   umode + rv_d
ENTRY   kmode + rv_d

/*

Although the location of the operand must be tested for being cached as
in the previous cases, a further problem arises with rv_d since it is
```

possible for the most significant word to be in the cache (at the cache base) and the LSW to be swapped out into main memory. As the cache base lies on a double word boundary this can only happen if the address is odd. If the address is even and is in main memory a double word read operation can be performed.

The code at @d1 and @d2 is reached if main memory must be read, the first instruction tests for a read fault on fetching the LSW. If the address is even a CJV is taken. Otherwise, another memory reference is started. At this point it is determined whether the MSW lies at the same address as the cache base and if so the code at @d8 is reached which loads the word from the cache on to the stack.

```
*/

        CONT    DZ D=CSH  OR  RAMF B=R0  LVAR  LDCA
        CJP     NBW, @d1        AB A=R0 B=cb  SUBR
        CJP     NBW, @d2        AB A=R0 B=sp  SUB  INCCA  D=TB  LBR  ADDR
        CJS     RFLT, rcache  DZ D=CAIR MASK  OR  RAMF B=R2  LOCK
        LDCT    @fixup  DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR  LOCK
        CONT    DZ D=CSH  OR  RAMF B=R0  DECCA
        CONT    ZA  ADD CIN  RAMA A=sp B=sp  LDCA
        CJV     CSH CWR  INCCA
        CONT    R0 CWR
//----------

@d1:
        CJS     RFLT, rcache  DZ D=CAIR MASK  OR  RAMF B=R2  LOCK
@d2:
        LDCT    @fixup  DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR  LRD
        CONT    ZB  ADD CIN RAMA A=sp B=sp  LDCA  RRD
        CJV     EV  ZA A=R0  OR  D=BUS  CWR  INCCA
        CONT    ZA A=R0  ADD CIN  LVAR  D=BUS  CWR

        CJP     Z, @d8  AB A=R0 B=cb  SUBR  D=TB  LBR  ADDR
        CJS     RFLT, rcache  ZB  SUBR  RAMF B=sp  LDCA  LOCK
        LDCT    @fixup  RD
        CJV     ZB  ADD CIN  RAMF B=sp  INCCA
        CONT    D=BUS  CWR
//----------

@d8:
        LDCT    @fixup  cb  LDCA  LOCK
        CJV     ZB  ADD CIN RAMF B=sp  LDCA
        CONT    CSH  CWR
//---------


ENTRY   umode + rv_b
ENTRY   kmode + rv_b

/*
```

The TOS now contains the mask (see lvb.m for a description of the bit manipulation required); this is read into R0 on the first cycle, simultaneously reloading the IR and decrementing CA to access the pointer. While waiting for the cache the stack pointer is decremented. At this point the context is almost identical to that in lvb_b and very similar code is used, the differences being that the address to be loaded in the VAR is coming straight from the cache and that the mask is already in the cache. The other difference should never be detected - if the mask is zero then the value left on the stack is the address rather than the mask.

```
*/

        LDCT    rcache  DZ D=CSH  OR  RAMA A=ir0 B=R0  LDIR  DECCA
        CONT    ZB  SUBR  RAMF B=sp
        CONT    DZ D=CSH  OR  RAMF B=R2  LVAR
        CJP     NBW, @b1        AB A=cb B=R2  SUB  D=TB  LBR  ADDR
        CJP     NBW, @b2        AB A=sp B=R2  SUBR  LOCK
        CJP     Z, R0, @b5  RD
// Cache here
        CONT    DZ SHR1  OR  RAMA A=ir0 B=ir0
        CONT    R2  LDCA
        CJP     @b3     sp  LDCA
        CONT    DZ D=CSH  OR  RAMU ROT B=R1
@b1:
        CJP     Z, R0, @b5  RD
@b2:
        CJP     RFLT, @b4       DZ SHR1  OR  RAMA A=ir0 B=ir0
        CONT    DZ D=BUS  OR  RAMU ROT B=R1
@b3:
                CJP     EV, $   ZB  OR  RAMD ZERO B=R0
@b5:
                CONT    ZB  OR  RAMD ROT B=R1
        CJV     ZB  OR  RAMU ONE B=R0
        CONT    AB A=R0 B=R1  AND CWR
//---------

@b4:
        JSRP    F  DZ D=BR, rv_b  OR  RAMF B=R2
        LDCT    @fixup  ZB  ADD CIN RAMF B=sp  INCCA
        CJP     @b3     ZB  SUBR  RAMF B=sp  DECCA
        CONT    DZ D=BUS  OR  RAMU  ROT B=R1
//----------
```

```
/*
        i_b_ari  --  Arithmetic and bitwise binary operators - integers

        Copyright (c) 1984 by High Level Hardware Limited

*/

*I      register.m
*I      opcodes.m
*I      trapcodes.m

EXTERNAL        tcdivcore, multiply, trap
GLOBAL          int_ovflo


ENTRY   umode + mul_i
ENTRY   kmode + mul_i

/*

mul_i replaces TOS and NOS by NOS*TOS, where the values are two's
complement 32 bit integers. An external, general purpose, 2's complement
multiplication subroutine is used. This routine multiplies R0 by Q,
returning the 64 bit product in R1 Q and preserves R0. First see if
Integer Arithmetic Traps Enable bit of psw is set (0B), in which case
check for overflows after the multiplication. The more significant word
is returned in R1; if there is no overflow it will be the sign extension
of Q. If an exception is detected we must trap if IATE is set. Note the
compact code where the condition is tested at @neg or at @pos.

*/

        CJP     OB, @excpt      DZ D=CSH  OR  RAMA A=psw B=R0  DECCA
        CJS     multiply        DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
        CONT    DZ D=CSH  OR  QREG

        CJV     ZQ  OR  CWR
        CONT    ZB  SUBR  RAMF B=sp
//----------

@excpt:         // IATE set, check for exceptions.

        CONT    DZ D=CSH  OR  QREG

        CJP     S,@neg  ZQ  OR  CWR
        JUMP    $+2     ZB  SUBR  RAMF B=sp
@pos:
        CJV     Z       ZA A=R1  OR     // R1 must be zero if Q > 0.
        CONT    DZ D=CAIR MASK  OR  QREG

        CONT    DZ D=BR,mul_i  OR  LDIR
        JUMP    int_ovflo       DQ SHL1  OR  RAMA A=ir0 B=ir0
        CONT    S, R1                   // Underflow if negative.
```

```
//----------

@neg:
        CJV     Z       ZA A=R1  EXNOR  // R1 must be -1 if Q < 0.
//----------

ENTRY   umode + div_i
ENTRY   kmode + div_i

/*

div_i replaces TOS and NOS by NOS/TOS, where the values are two's
complement 32 bit integers. An external, general purpose, 2's complement
division subroutine is used. On the first instruction the divisor is
read and placed in R0 ready for the division. If the divisor is zero the
instruction is aborted and a meaningless answer returned from the code at
@4 if the IDBZE bit of psw is zero, otherwise a trap is generated. The
dividend is read next (into QREG) and sign extended to 64 bits, using R1
as the most significant words. Sign extension occurs on the first LDCT
if positive and at @3 if negative. The quotient is returned in QREG, but
may need some slight modification. Much of the rest of this segment of
code is concerned with preparing for the division routine proper and
tidying up afterwards - an imperfectly understood procedure but it seems
to work!

*/

        CJP     Z, @dbz_d       DZ CSH  OR  RAMF B=R0  DECCA
        CONT    DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR

        CJP     S, @3   DZ CSH  OR  QREG
        CJS     tcdivcore       ZA A=R0  OR  RAMQU B=R1  TCDIVF

        LDCT, 14        ZB  AND  RAMF B=R1  TCDIV
        CJP     S, @2   DA CSH A=R0  ADD  RAMF B=R1
        CJV     ZB  SUBR  RAMF B=sp
        CONT    ZQ  CWR
@1:
        CJV     ZB  SUBR  RAMF B=sp
@2:
        CONT    ZQ  ADD  CIN  CWR
//----------

@3:
        LDCT, 14        AB A=R1  SUBR  RAMF B=R1  TCDIV
        CONT    DA CSH A=R0  ADD  RAMF B=R1
        CJP     S, @1   ZB  SUB CIN  B=R1
        CJP     Z, @2   AB  A=R0 B=R1  SUBR CIN
//----------

@4:
        CJV     ZB  SUBR  RAMF B=sp
        CONT    ZQ  OR  CWR
```

```
//----------

@dbz_d:
        CONT    DZ SHR3  OR  RAMA A=psw B=R0
        CJV     EV      ZB  SUBR  RAMA A=R0 B=sp
        CJP     NLC, @dbz       DZ D=CAIR MASK  OR  QREG
        CONT    DZ D=BR,div_i  OR  LDIR
//----------

/*

mod_i replaces TOS and NOS by NOS%TOS, where the values are two's
complement 32 bit integers.  The same division routine is used as for
div_i the remainder being returned in R1 (subject to small
modifications).  Otherwise this operation is analogous to div_i.

*/

ENTRY   umode + mod_i
ENTRY   kmode + mod_i

        CJP     Z, @dbz_m       DZ CSH  OR  RAMF B=R0  DECCA
        CONT    DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
        CJP     S, @7   DZ CSH  OR  QREG
        CJS     tcdivcore       ZA A=R0  OR  RAMQU B=R1  TCDIVF
        LDCT, 14        ZB  AND  RAMF B=R1 TCDIV
        CJP     S, @6   DA CSH A=R0  ADD  RAMF B=R1
        CJV     ZB  SUBR  RAMF B=sp
        CONT    R1  CWR
//----------

@5:
        CJV     ZB  SUBR  RAMF B=sp
@6:
        CONT    AB A=R0 B=R1  SUBR CIN  CWR
//----------

@7:

        LDCT, 14        AB A=R1  SUBR  RAMF B=R1  TCDIV
        CONT    DA CSH A=R0  ADD  RAMF B=R1
        CJP     Z, @5   AB A=R0 B=R1  SUBR CIN
        CJP     S, @6   ZB  SUB CIN  B=R1
//----------

        CJV     ZB  SUBR  RAMF B=sp
@8:
        CONT    R1  CWR
//----------

@dbz_m:
        CONT    DZ SHR3  OR  RAMA A=psw B=R0
        CJV     EV      ZB  SUBR  RAMA A=R0 B=sp
```

```
        CONT    DZ D=CAIR MASK  OR  QREG
        CONT    DZ D=BR,mod_i  OR  LDIR
@dbz:
        JUMP    trap    DQ SHL1  OR  RAMA A=ir0 B=ir0
        CONT    DZ D=BR, int_dbz  OR  QREG
//----------
```

```
/*

The following group of operators all have the same structure and  replace
NOS and TOS by NOS <op> TOS.  The instruction register is reloaded on the
first instruction, the value at TOS is loaded into R0 on the second.  The
other  operand  is read on the third instruction, the operation performed
and written back to R0.  The final instruction writes the result  to  the
cache and decrements the stack pointer.

*/
```

```
ENTRY   umode + add_i           // NOS + TOS
ENTRY   kmode + add_i

        CONT    DZ SHR1  OR  RAMA A=ir0 B=ir0  DECCA  LDIR
        CONT    DZ CSH  OR  RAMF B=R0
        CJV     NO      DA CSH A=R0  ADD  RAMF B=R0
        CONT    ZB  SUBR  RAMA A=R0 B=sp  CWR

        CJV     EB, psw
        CJP     NLC, @over      DZ D=CAIR MASK  OR  QREG
        CONT    DZ D=BR,add_i  OR  LDIR
//----------

ENTRY   umode + sub_i           // NOS - TOS
ENTRY   kmode + sub_i

        CONT    DZ SHR1  OR  RAMA A=ir0 B=ir0  DECCA  LDIR
        CONT    DZ CSH  OR  RAMF B=R0
        CJV     NO      DA CSH A=R0  SUB CIN  RAMF B=R0
        CONT    ZB  SUBR  RAMA A=R0 B=sp  CWR

        CJV     EB, psw
        CONT    DZ D=CAIR MASK  OR  QREG
        CONT    DZ D=BR,sub_i  OR  LDIR

@over:
        JUMP    int_ovflo       DQ SHL1  OR  RAMA A=ir0 B=ir0
        CONT    NS, R1
//----------


ENTRY   umode + and_i           // NOS & TOS
ENTRY   kmode + and_i
```

```
        CONT    DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR  DECCA
        JUMP    @9      DZ CSH  OR  RAMF B=R0
        CJV     DA CSH A=R0  AND  RAMF B=R0
//----------

ENTRY   umode + xor_i           // NOS ^ TOS
ENTRY   kmode + xor_i

        CONT    DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR  DECCA
        JUMP    @9      DZ CSH  OR  RAMF B=R0
        CJV     DA CSH A=R0  EXOR  RAMF B=R0
//----------

ENTRY   umode + or_i            // NOS | TOS
ENTRY   kmode + or_i

        CONT    DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR  DECCA
        JUMP    @9      DZ CSH  OR  RAMF B=R0
        CJV     DA CSH A=R0  OR  RAMF B=R0
@9:
        CONT    ZB  SUBR  RAMA A=R0 B=sp  CWR
//----------

/*

rshift_i and lshift_i differ only in that the former shifts the int right
(towards the least significant bit), bringing in a copy of the most
significant (sign) bit whilst the latter shifts to the left, bringing in
zero. The value at TOS is the number of places to shift NOS, this is
read in, stored in R1 (actually -count is stored) and checked that it is
greater than zero. If not the operation is aborted, in this case we
return NOS as the result. Since it is a waste of time to shift more than
32 places a 32 iteration loop is set up on the following operation; the
instruction register and ir0 are also reloaded. A TWB sequencer
instruction is used to terminate the loop when either the count in R1
becomes zero or when 32 iterations have occurred, whichever comes first.
The result is written to the cache and the stack pointer decremented
immediately the loop finishes. If the instruction is aborted then,
because of the pipe-line delay, the sequencer stack has been PUSHed and
must be popped before passing control to the next operation.

*/

ENTRY   umode + lshift_i
ENTRY   kmode + lshift_i

        CJP     NS, @abort      DZ CSH  SUBR CIN  RAMF B=R1  DECCA
        PUSH, 31        DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
        CONT    DZ CSH  OR  RAMF B=R0

                TWB     Z, @10  ZA A=R1  ADD CIN  RAMF B=R1
                CONT    ZA A=R0  OR  RAMU ZERO B=R0
@10:
```

```
        CJV     ZB  SUBR  RAMA A=R0 B=sp  CWR
        CONT
//----------

ENTRY   umode + rshift_i
ENTRY   kmode + rshift_i

        CJP     NS, @abort      DZ CSH  SUBR CIN  RAMF B=R1  DECCA
        PUSH, 31        DZ SHR1  OR  RAMA A=ir0 B=ir0  LDIR
        CONT    DZ CSH  OR  RAMF B=R0

                TWB     Z, @11  ZA A=R1  ADD CIN  RAMF B=R1
                CONT    ZA A=R0  OR  RAMD ARI B=R0

@11:
        CJV     ZB  SUBR  RAMA A=R0 B=sp  CWR
        CONT
//----------

@abort:
        CJV     ZB  SUBR  RAMF B=sp
        LOOP
//----------

int_ovflo:

/*

General purpose selector for overflow or underflow trap codes.  Jump into
it with CC set to TRUE for underflow, FALSE for overflow.

*/

        CJP     LC, @under
        JUMP    trap
        CONT    DZ D=BR,int_overflow  OR  QREG
@under:
        CONT    DZ D=BR,int_underflow  OR  QREG
//----------
```

```
/*
        muldiv  --  Multiplication and division subroutines

        Copyright (c) 1983 by High Level Hardware Limited

*/

*I      register.m

GLOBAL  multiply, usmultiply, tcdivcore, usdivide

multiply:

/*

Two's complement multiply of R0 by QREG, giving answer in R1 QREG and
preserving R0. R1 is initially cleared and the sequencer counter loaded
to give 15 iterations of the loop. Since the RPCT controlled loop
contains two microinstructions this gives 30 operations of the 32
necessary. At each stage the least significant bit of QREG is shifted
out and a MUL special function specified so that if the shifted-out bit
is zero then on the next cycle the AB ALU source specification is changed
to ZB. The final instructions perform the two remaining shifts and
return to the calling routine. The last operation is a conditional
subtract since the most significant bit in a two's complement number
carries negative weight.

*/

        LDCT, 14        ZA  AND  RAMQD ZERO B=R1  MUL

                RPCT $  AB A=R0  ADD  RAMQD TC B=R1  MUL
                CONT    AB A=R0  ADD  RAMQD TC B=R1  MUL

        CRTN    AB A=R0  ADD  RAMQD TC B=R1  MUL
        CONT    AB A=R0 SUBR CIN RAMQD TC B=R1
//----------

usmultiply:

/*

Unsigned multiply is identical to the signed case except that a different
bit is shifted in, denoted by US rather than TC.

*/

        LDCT, 14        ZB  AND  RAMQD ZERO B=R1  MUL

                RPCT $  AB A=R0  ADD  RAMQD US B=R1  MUL
                CONT    AB A=R0  ADD  RAMQD US B=R1  MUL

        CRTN    AB A=R0  ADD  RAMQD US B=R1  MUL
```

```
        CONT    AB A=R0  ADD  RAMQD US B=R1
//----------

tcdivcore:

/*

This amazing code performs the nub of a two's complement non-restoring
division. The 64 bit number in R1 QREG is divided by R0, giving a
quantity close to the remainder in R1 and sometimes giving the quotient
in QREG. The detailed working of the routine is not understood but it
always works. The TOS is used as temporary storage.

See div_i and mod_i for examples of how to call this routine and how to
interpret the results.

*/

                RPCT $  AB A=R0  ADD  RAMQU TC B=R1  TCDIV
                CONT    AB A=R0  ADD  RAMQU TC B=R1  TCDIV

        CRTN    AB A=R0  ADD  RAMQU TC B=R1  TCDIV
        CONT    AB A=R0  ADD  RAMQU TC B=R1  DIVL  CWR
//----------

usdivide:

/*

Much the same must be said for this unsigned division routine.

See div_u and mod_u for examples of use.

*/

        LDCT, 15        USDIVF

                RPCT $  AB A=R0  ADD  RAMQU US B=R1  USDIV
                CONT    AB A=R0  ADD  RAMQU US B=R1  USDIV

        CRTN    AB A=R0  ADD  RAMQU US B=R1  DIVL  CWR
        CONT    DA CSH A=R0  ADD  RAMF B=R1
//----------
```

```
/*
        unimp  --  Unimplemented instruction trap

        Copyright (c) 1983 by High Level Hardware Limited
*/

*I      opcodes.m
*I      trapcodes.m

EXTERNAL        trap

DEFAULTENTRY    umode
DEFAULTENTRY    aumode
DEFAULTENTRY    kmode
DEFAULTENTRY    akmode

DEFAULTENTRY    clmode
DEFAULTENTRY    aclmode


// Jump to unimplemented instruction trap and load QREG with the trap code.

ENTRY   aumode + esc    // esc esc is always unimplemented.
ENTRY   apmode + esc
ENTRY   akmode + esc

        JUMP    trap
        CONT    DZ D=BR, unimplemented  OR  QREG
//----------
```

```
/*
        profile  --  Dynamic profiler and single step mechanism

        Copyright (c) 1984 by High Level Hardware Limited
*/

*I      register.m
*I      opcodes.m
*I      trapcodes.m

EXTERNAL        rwcache, trap, noop

/*

We assume that register pr has been set by the operating system to
contain the virtual address of a profiling area. psw contains the
current instruction set number in bits 8-11. Profiling is enabled by the
operating system setting the profile enable bit in psw, with rei being
responsible for setting HIR to be 'pinst'. Single stepping is enabled by
setting bit 14 in psw. The operating system arranges that single
stepping and profiling are never simultaneously enabled.

*/

DEFAULTENTRY    pmode

/*

Enter here after an LDIR. First test for single stepping. If we are, be
careful to preserve VAR then transfer to sstep. Otherwise, add the
contents of IR to the profile base register (pr) and increment the
contents of the virtual location pointed to. The current instruction set
is taken from bits 8-11 of psw.

*/

        CJP     NS, sstep       DZ RTL2  OR  RAMA A=psw B=psw
        CJP     NLC,@1
        CONT    DA D=CAIR ZZZD A=pr  ADD  LVAR
//----------


DEFAULTENTRY    apmode

/*

Enter here after an ALDIR. Add the contents of IR + 256 to profile base
register (pr) and increment the contents of the virtual location pointed
to. The current instruction set is taken from bits 8-11 of psw.

*/

        CJP     NS, sstep       DZ RTL2  OR  RAMA A=psw B=psw
```

```
        CONT    DA D=CAIR ZZZD A=pr  ADD  RAMF B=R0
        CONT    DA D=BR,256 A=R0  ADD  LVAR
@1:
        CONT    D=TB  LBR  ADDR
        CJS     WFLT, rwcache   DZ ZZZA  OR  RAMA A=psw B=R0  LOCK
        LDCT    noop  RD        DZ RTR2  OR  RAMA A=psw B=psw
        CONT    DZ  D=BR, pinst  OR  RAMA A=R0 B=R0  HLDIR LOCK
        CJV     DZ D=BUS  ADD CIN  RAMA A=R0 B=R0  HLDIR  LOCK
        CONT    R0  LBR  WR
//----------


sstep:

/*

Single stepping here.  Trap  with  the  single  stepping  trapcode  after
restoring the psw and pushing the IR back into the code stream.

*/

        CONT    DZ RTR2  OR  RAMA A=psw B=psw
        CONT    DZ SHL1  OR  RAMA A=ir0 B=ir0
        JUMP    trap  DA D=CAIR ZZZD A=ir0  OR  RAMF B=ir0
        CONT    DZ D=BR,single_step  OR  QREG
//----------
```

APPENDIX F

A Practical Example

In this appendix we illustrate the steps involved in adding a new instruction to the standard system. Our example instruction, lppp_w, is similar to the lp_w instruction described in Chapter 10 and Appendix E except that the local variable is post incremented as well as being loaded onto the top of the stack. This instruction could well be generated by the C compiler for a fragment such as

```
{
        int i, *p;
            .
            .
            .
        i = *p++;
            .
            .
}
```

After compilation, the assignment statement could be implemented in assembly language as

```
lppp_w 4        # Load and increment p
rv_w            # Indirect
sp3             # Store in i
```

The microcode to implement the lppp_w instruction might be

```
*I register
*I opcodes

ENTRY umode + lppp_w

        CONT   DZ MASK  OR  RAMA A=ir0 B=R0  PLDIR
        CONT   AB A=fp B=R0  ADD  LDCA
        CONT   DZ SHR2  OR  RAMA A=ir0 B=ir0
        CONT   DZ D=CSH  ADD CIN  RAMF B=R0
        CONT   ZA A=R0  OR  CWR
        CJV    ZB  ADD CIN  RAMF B=sp  LDCA
        CONT   ZA A=R0  SUBR  CWR
//----------
```

Assuming that this source microcode is contained in the file lppp.m, and that the file opcodes.m contains a definition of the symbol lppp_w then the following commands typed to OTS will assemble, link, and load the code. First, we assemble lppp.m to produce lppp.mrl, a relocatable object module

```
ms lppp
```

Next we link this module to form an absolute load module lppp.mcd. In

this case, since this a self contained section of microcode, no other modules are required. We must, however, arrange that after linking the absolute image will be loaded into an unused area of the control store. In general this will depend on what other microcode is in use but we assume here that locations f00 (hex) and above are available.

```
ml -cf00 lppp
```

Finally, we load the code into the control store

```
loadmc lppp
```

The new instruction can now be tested by writing suitable high level programs.