DESCRIPTION OF A MACHINE ARCHITECTURE FOR USE WITH BLOCK STRUCTURED LANGUAGES

Andrew S. Tanenbaum
Hans van Staveren
Ed G. Keizer
Johan W. Stevenson[*]

August 1983

Informatica Rapport IR-81

Abstract

EM is a family of intermediate languages designed for producing portable compilers. A program called **front end** translates source programs to EM. Another program, **back end**, translates EM to the assembly language of the target machine. Alternatively, the EM program can be assembled to a highly efficient binary format for interpretation. This document describes the EM languages in detail.

[*] Present affiliation: NV Philips, Eindhoven

1. INTRODUCTION

EM is a family of intermediate languages designed for producing portable compilers. The general strategy is for a program called **front end** to translate the source program to EM. Another program, **back end** translates EM to target assembly language. Alternatively, the EM code can be assembled to a binary form and interpreted. These considerations led to the following goals:

1 The design should allow translation to, or interpretation on, a wide range of existing machines. Design decisions should be delayed as far as possible and the implications of these decisions should be localized as much as possible.
The current microcomputer technology offers 8, 16 and 32 bit machines with various sizes of address space. EM should be flexible enough to be useful on most of these machines. The differences between the members of the EM family should only concern the wordsize and address space size.

2 The architecture should ease the task of code generation for high level languages such as Pascal, C, Ada, Algol 68, BCPL.

3 The instruction set used by the interpreter should be compact, to reduce the amount of memory needed for program storage, and to reduce the time needed to transmit programs over communication lines.

4 It should be designed with microprogrammed implementations in mind; in particular, the use of many short fields within instruction opcodes should be avoided, because their extraction by the microprogram or conversion to other instruction formats is inefficient.

The basic architecture is based on the concept of a stack. The stack is used for procedure return addresses, actual parameters, local variables, and arithmetic operations. There are several built-in object types, for example, signed and unsigned integers, floating point numbers, pointers and sets of bits. There are instructions to push and pop objects to and from the stack. The push and pop instructions are not typed. They only care about the size of the objects. For each built-in type there are reverse Polish type instructions that pop one or more objects from the top of the stack, perform an operation, and push the result back onto the stack. For all types except pointers, these instructions have the object size as argument.

There are no visible general registers used for arithmetic operands etc. This is in contrast to most third generation computers, which usually have 8 or 16 general registers. The decision not to have a group of general registers was fully intentional, and follows W.L. Van der Poel's dictum that a machine should have 0, 1, or an infinite number of any feature. General registers have two primary uses: to hold intermediate results of complicated expressions, e.g.

$$((a*b + c*d)/e + f*g/h) * i$$

and to hold local variables.

Various studies have shown that the average expression has fewer than two operands, making the former use of registers of doubtful value. The present trend toward structured programs consisting of many small procedures greatly reduces the value of registers to hold local variables because the large number of procedure calls implies a large overhead in saving and restoring the registers at every call.

Although there are no general purpose registers, there are a few internal registers with specific functions as follows:

PC	−	Program Counter	Pointer to next instruction
LB	−	Local Base	Points to base of the local variables in the current procedure.
SP	−	Stack Pointer	Points to the highest occupied word on the stack.
HP	−	Heap Pointer	Points to the top of the heap area.

Furthermore, reverse Polish code is much easier to generate than multi-register machine code, especially if highly efficient code is desired. When translating to assembly language the back end can make good use of the target machine's registers. An EM machine can achieve high performance by keeping part of the stack in high speed storage (a cache or microprogram scratchpad memory) rather than in primary memory.

Again according to van der Poel's dictum, all EM instructions have zero or one argument. We believe that instructions needing two arguments can be split into two simpler ones. The simpler ones can probably be used in other circumstances as well. Moreover, these two instructions together often have a shorter encoding than the single instruction before.

This document describes EM at three different levels: the abstract level, the assembly language level and the machine language level.

The most important level is that of the abstract EM architecture. This level deals with the basic design issues. Only the functional capabilities of instructions are relevant, not their format or encoding. Most chapters of this document refer to the abstract level and it is explicitly stated whenever another level is described.

The assembly language is intended for the compiler writer. It presents a more or less orthogonal instruction set and provides symbolic names for data. Moreover, it facilitates the linking of separately compiled 'modules' into a single program by providing several pseudoinstructions.

The machine language is designed for interpretation with a compact program text and easy decoding. The binary representation of the machine language instruction set is far from orthogonal. Frequent instructions have a short opcode. The encoding is fully byte oriented. These bytes do not contain small bit fields, because bit fields would slow down decoding considerably.

A common use for EM is for producing portable (cross) compilers. When used this way, the compilers produce EM assembly language as their output. To run the compiled program on the target machine, the back end, translates the EM assembly language to the target machine's assembly language. When this approach is used, the format of the EM machine language instructions is irrelevant. On the other hand, when writing an interpreter for EM machine language programs, the interpreter must deal with the machine language and not with the symbolic assembly language.

As mentioned above, the current microcomputer technology offers 8, 16 and 32 bit machines with address spaces ranging from 2^{16} to 2^{32} bytes. Having one size of pointers and integers restricts the usefulness of the language. We decided to have a different language for each combination of word and pointer size. All languages offer the same instruction set and differ only in memory alignment restrictions and the implicit size assumed in several instructions. The languages differ slightly for the different size combinations. For example: the size of any object on the stack and alignment restrictions. The wordsize is restricted to powers of 2 and the pointer size must be a multiple of the wordsize. Almost all programs handling EM will be parametrized with word and pointer size.

2. MEMORY

The EM machine has two distinct address spaces, one for instructions and one for data. The data space is divided up into 8-bit bytes. The smallest addressable unit is a byte. Bytes are numbered consecutively from 0 to some maximum. All sizes in EM are expressed in bytes.

Some EM instructions can transfer objects containing several bytes to and/or from memory. The size of all objects larger than a word must be a multiple of the wordsize. The size of all objects smaller than a word must be a divisor of the wordsize. For example: if the wordsize is 2 bytes, objects of the sizes 1, 2, 4, 6,... are allowed. The address of such an object is the lowest address of all bytes it contains. For objects smaller than the wordsize, the address must be a multiple of the object size. For all other objects the address must be a multiple of the wordsize. For example, if an instruction transfers a 4-byte object to memory at location m and the wordsize is 2, m must be a multiple of 2 and the bytes at locations m, $m+1$,$m+2$ and $m+3$ are overwritten.

The size of almost all objects in EM is an integral number of words. Only two operations are allowed on objects whose size is a divisor of the wordsize: push it onto the stack and pop it from the stack. The addressing of these objects in memory is always indirect. If such a small object is pushed onto the stack it is assumed to be a small integer and stored in the least significant part of a word. The rest of the word is cleared to zero, although EM provides a way to sign-extend a small integer. Popping a small object from the stack removes a word from the stack, stores the least significant byte(s) of this word in memory and discards the rest of the word.

The format of pointers into both address spaces is explicitly undefined. The size of a pointer, however, is fixed for a member of EM, so that the compiler writer knows how much storage to allocate for a pointer.

A minor problem is raised by the undefined pointer format. Some languages, notably Pascal, require a special, otherwise illegal, pointer value to represent the nil pointer. The current Pascal-VU compiler uses the integer value 0 as nil pointer. This value is also used by many C programs as a normally impossible address. A better solution would be to have a special instruction loading an illegal pointer value, but it is hard to imagine an implementation for which the current solution is inadequate, especially because the first word in the EM data space is special and probably not the target of any pointer.

The next two chapters describe the EM memory in more detail. One describes the instruction address space, the other the data address space.

A design goal of EM has been to allow its implementation on a wide range of existing machines, as well as allowing a new one to be built in hardware. To this extent we have tried to minimize the demands of EM on the memory structure of the target machine. Therefore, apart from the logical partitioning, EM memory is divided into 'fragments'. A fragment consists of consecutive machine words and has a base address and a size. Pointer arithmetic is only defined within a fragment. The only exception to this rule is comparison with the null pointer. All fragments must be word aligned.

3. INSTRUCTION ADDRESS SPACE

The instruction space of the EM machine contains the code for procedures. Tables necessary for the execution of this code, for example, procedure descriptor tables, may also be present. The instruction space does not change during the execution of a program, so that it may be protected. No further restrictions to the instruction address space are necessary for the abstract and assembly language level.

Each procedure has a single entry point: the first instruction. A special type of pointer identifies a procedure. Pointers into the instruction address space have the same size as pointers into data space and can, for example, contain the address of the first instruction or an index in a procedure descriptor table.

There is a single EM program counter, PC, pointing to the next instruction to be executed. The procedure pointed to by PC is called the ’current’ procedure. A procedure may call another procedure using the CAL or CAI instruction. The calling procedure remains ’active’ and is resumed whenever the called procedure returns. Note that a procedure has several ’active’ invocations when called recursively.

Each procedure must return properly. It is not allowed to fall through to the code of the next procedure. There are several ways to exit from a procedure:

- the RET instruction, which returns to the calling procedure.
- the RTT instruction, which exits a trap handling routine and resumes the trapping instruction (see next chapter).
- the GTO instruction, which is used for non-local goto’s. It can remove several frames from the stack and transfer control to an active procedure. (see also MES 11 in paragraph 11.1.4.4)

All branch instructions can transfer control to any label within the same procedure. Branch instructions can never jump out of a procedure.

Several language implementations use a so called procedure instance identifier, a combination of a procedure identifier and the LB of a stack frame, also called static link.

The program text for each procedure, as well as any tables, are fragments and can be allocated anywhere in the instruction address space.

4. DATA ADDRESS SPACE

The data address space is divided into three parts, called ’areas’, each with its own addressing method: global data area, local data area (including the stack), and heap data area. These data areas must be part of the same address space because all data is accessed by the same type of pointers.

Space for global data is reserved using several pseudoinstructions in the assembly language, as described in the next paragraph and chapter 11. The size of the global data area is fixed per program.

Global data is addressed absolutely in the machine language. Many instructions are available to address global data. They all have an absolute address as argument. Examples are LOE, LAE and STE.

Part of the global data area is initialized by the compiler, the rest is not initialized at all or is initialized with a value, typically −32768 or 0. Part of the initialized global data may be made read-only if the implementation supports protection.

The local data area is used as a stack, which grows from high to low addresses and contains some data for each active procedure invocation, called a ’frame’. The size of the local data area varies dynamically during execution. Below the current procedure frame resides the operand stack. The stack pointer SP always points to the bottom of the local data area. Local data is addressed by offsetting from the local base pointer LB. LB always points to the frame of the current procedure. Only the words of the current frame and the parameters can be addressed directly. Variables in other active procedures are addressed by following the chain of statically enclosing procedures using the LXL or LXA instruction. The variables in dynamically enclosing procedures can be addressed with the use of the DCH instruction.

Many instructions have offsets to LB as argument, for instance LOL, LAL and STL. The arguments of these instructions range from −1 to some (negative) minimum for the access of local storage and from 0 to some (positive) maximum for parameter access.

The procedure call instructions CAL and CAI each create a new frame on the stack. Each procedure has an assembly-time parameter specifying the number of bytes needed for local storage. This storage is allocated each time the procedure is called and must be a multiple of the wordsize. Each procedure, therefore, starts with a stack with the local variables already allocated. The return instructions RET and RTT remove a frame. The actual parameters must be removed by the calling procedure.

RET may copy some words from the stack of the returning procedure to an unnamed ’function return area’. This area is available for ’READ-ONCE’ access using the LFR instruction. The result of a LFR is only defined if the size used to fetch is identical to the size used in the last return. The instruction ASP, used to remove the parameters from the stack, the branch instruction BRA and the non-local goto instruction GTO are the only ones that leave the contents of the ’function return area’ intact. All other instructions are allowed to destroy the function return area. Thus parameters can be popped before fetching the function result. The maximum size of all function return areas is implementation dependent, but should allow procedure instance identifiers and all implemented objects of type integer, unsigned, float and pointer to be returned. In most implementations the maximum size of the function return area is twice the pointer size, because we want to be able to handle ’procedure instance identifiers’ which consist of a procedure identifier and the LB of a frame belonging to that procedure.

The heap data area grows upwards, to higher numbered addresses. It is initially empty. The initial value of the heap pointer HP marks the low end. The heap pointer may be manipulated by the LOR and STR instructions. The heap can only be addressed indirectly, by pointers derived from previous values of HP.

4.1 Global data area

The initial size of the global data area is determined at assembly time. Global data is allocated by several pseudoinstructions in the EM assembly language. Each pseudoinstruction allocates one or more bytes. The bytes allocated for a single pseudo form a ’block’. A block differs from a fragment, because, under certain conditions, several blocks are allocated in a single fragment. This guarantees that the bytes of these blocks are consecutive.

Global data is addressed absolutely in binary machine language. Most compilers, however, cannot assign absolute addresses to their global variables, especially not if the language allows programs to be composed of several separately compiled modules. The assembly language therefore allows the compiler to name the first address of a global data block with an alphanumeric label. Moreover, the only way to address such a named global data block in the assembly language is by using its name. It is the task of the assembler/loader to translate these labels into absolute addresses. These labels may also be used in CON and ROM pseudoinstructions to initialize pointers.

The pseudoinstruction CON allocates initialized data. ROM acts like CON but indicates that the initialized data will not change during execution of the program. The pseudoinstruction BSS allocates a block of uninitialized or identically initialized data. The pseudoinstruction HOL is similar to BSS, but it alters the meaning of subsequent absolute addressing in the assembly language.

Another type of global data is a small block, called the ABS block, with an implementation defined size. Storage in this type of block can only be addressed absolutely in assembly language. The first word has address 0 and is used to maintain the source line number. Special instructions LIN and LNI are provided to update this counter. A pointer at location 4 points to a string containing the current source file name. The instruction FIL can be used to update the pointer.

All numeric arguments of the instructions that address the global data area refer to locations in the ABS block unless they are preceded by at least one HOL pseudo in the same module, in which case they refer to the storage area allocated by the last HOL pseudoinstruction. Thus LOE 0 loads the zeroth word of the most recent HOL, unless no HOL has appeared in the current file so far, in which case it loads the zeroth word of the ABS fragment.

The global data area is highly fragmented. The ABS block and each HOL and BSS block are separate fragments. The way fragments are formed from CON and ROM blocks is more complex. The assemblers group several blocks into a single fragment. A fragment only contains blocks of the same type: CON or ROM. It is guaranteed that the bytes allocated for two consecutive CON pseudos are allocated consecutively in a single fragment, unless these CON pseudos are separated in the assembly language program by a data label definition or one or more of the following pseudos:

ROM, BSS, HOL and END

An analogous rule holds for ROM pseudos.

4.2 Local data area

The local data area consists of a sequence of frames, one for each active procedure. Below the frame of the current procedure resides the expression stack. Frames are generated by procedure calls and are removed by procedure returns. A procedure frame consists of six ’zones’:

1. The return status block
2. The local variables and compiler temporaries
3. The register save block
4. The dynamic local generators
5. The operand stack.
6. The parameters of a procedure one level deeper

A sample frame is shown in Figure 1.

Before a procedure call is performed the actual parameters are pushed onto the stack of the calling procedure. The exact details are compiler dependent. EM allows procedures to be called with a variable number of parameters. The implementation of the C-language almost forces its runtime system to push the parameters in reverse order, that is, the first positional parameter last. Most compilers use the C calling convention to be compatible. The parameters of a procedure belong to the frame of the calling procedure. Note that the evaluation of the actual parameters may imply the calling of procedures. The parameters can be accessed with certain instructions using offsets of 0 and greater. The first byte of the last parameter pushed has offset 0. Note that the parameter at offset 0 has a special use in the instructions following the static chain (LXL and LXA). These instructions assume that this parameter contains the LB of the statically enclosing procedure. Procedures that do not have a dynamically enclosing procedure do not need a static link at offset 0.

Two instructions are available to perform procedure calls, CAL and CAI. Several tasks are performed by these call instructions.

First, a part of the status of the calling procedure is saved on the stack in the return status block. This block should contain the return address of the calling procedure, its LB and other implementation dependent data. The size of this block is fixed for any given implementation because the lexical instructions LPB, LXL and LXA must be able to obtain the base addresses of the procedure parameters **and** local variables. An alternative solution can be used on machines with a highly segmented address space. The stack frames need not be contiguous then and the first status save area can contain the parameter base AB, which has the value of SP just after the last parameter has been pushed.

Second, the LB is changed to point to the first word above the local variables. The new LB is a copy of the SP after the return status block has been pushed.

Third, the amount of local storage needed by the procedure is reserved. The parameters and local storage are accessed by the same instructions. Negative offsets are used for access to local variables. The highest byte, that is the byte nearest to LB, has to be accessed with offset −1. The pseudoinstruction specifying the entry point of a procedure, has an argument that specifies the amount of local storage needed. The local variables allocated by the CAI or CAL instructions are the only ones that can be accessed with a fixed negative offset. The initial value of the allocated words is not defined, but implementations that check for undefined values will probably initialize them with a special 'undefined' pattern, typically −32768.

Fourth, any EM implementation is allowed to reserve a variable size block beneath the local variables. This block could, for example, be used to save a variable number of registers.

Finally, the address of the entry point of the called procedure is loaded into the Program Counter.

The ASP instruction can be used to allocate further (dynamic) local storage. The base address of such storage must be obtained with a LOR SP instruction. This same instruction ASP may also be used to remove some words from the stack.

There is a version of ASP, called ASS, which fetches the number of bytes to allocate from the stack. It can be used to allocate space for local objects whose size is unknown at compile time, so called 'dynamic local generators'.

Control is returned to the calling procedure with a RET instruction. Any return value is then copied to the 'function return area'. The frame created by the call is deallocated and the status of the calling procedure is restored. The value of SP just after the return value has been popped must be the same as the value of SP just before executing the first instruction of this invocation. This means that when a RET is executed the operand stack can only contain the return value and all dynamically generated locals must be deallocated. Violating this restriction might result in hard to detect errors. The calling procedure has to remove the parameters from the stack. This can be done with the aforementioned ASP instruction.

Each procedure frame is a separate fragment. Because any fragment may be placed anywhere in memory, procedure frames need not be contiguous.

```
|================================|
|      actual parameter  n-1     |
|--------------------------------|
|               .                |
|               .                |
|               .                |
|--------------------------------|
|      actual parameter  0       | ( <- AB )
|================================|


|================================|
|////////////////////////////////|
|///// return status block /////|
|////////////////////////////////|   <- LB
|================================|
|                                |
|        local variables         |
|                                |
|--------------------------------|
|                                |
|      compiler temporaries      |
|                                |
|================================|
|////////////////////////////////|
|///// register save block /////|
|////////////////////////////////|
|================================|
|                                |
|   dynamic local generators     |
|                                |
|================================|
|            operand             |
|--------------------------------|
|            operand             |
|================================|
|          parameter  m-1        |
|--------------------------------|
|               .                |
|               .                |
|               .                |
|--------------------------------|
|          parameter  0          | <- SP
|================================|
```

Figure 1. A sample procedure frame and parameters.

4.3 Heap data area

The heap area starts empty, with HP pointing to the low end of it. HP always contains a word address. A copy of HP can always be obtained with the LOR instruction. A new value may be stored in the heap pointer using the STR instruction. If the new value is greater than the old one, then the heap grows. If it is smaller, then the heap shrinks. HP may never point below its original value. All words between the current HP and the original HP are allocated to the heap. The heap may not grow into a part of memory that is already allocated. When this is attempted, the STR instruction will cause a trap to occur. In this case, HP retains its old value.

The only way to address the heap is indirectly. Whenever an object is allocated by increasing HP, then the old HP value must be saved and can be used later to address the allocated object. If, in the meantime, HP is decreased so that the object is no longer part of the heap, then an attempt to access the object is not allowed. Furthermore, if the heap pointer is increased again to above the object address, then access to the old object gives undefined results.

The heap is a single fragment. All bytes have consecutive addresses. No limits are imposed on the size of the heap as long as it fits in the available data address space.

5. MAPPING OF EM DATA MEMORY ONTO TARGET MACHINE MEMORY

The EM architecture is designed to be implemented on many existing and future machines. EM memory is highly fragmented to make adaptation to various memory architectures possible. Format and encoding of pointers is explicitly undefined.

This chapter gives solutions to some of the anticipated problems. First, we describe a possible memory layout for machines with 64K bytes of address space. Here we use a member of the EM family with 2-byte word and pointer size. The most straightforward layout is shown in figure 2.

```
65534 -> |-------------------------------|
         |///////////////////////////////|
         |//// unimplemented memory /////|
         |///////////////////////////////|
   ML -> |-------------------------------|
         |                               |
         |                               | <- LB
         |     stack and local area      |
         |                               |
         |-------------------------------| <- SP
         |///////////////////////////////|
         |//////// inaccessible /////////|
         |///////////////////////////////|
         |-------------------------------| <- HP
         |                               |
         |           heap area           |
         |                               |
         |                               |
   HB -> |-------------------------------|
         |                               |
         |       global data area        |
         |                               |
   EB -> |-------------------------------|
         |                               |
         |         program text          | <- PC
         |                               |
         |        ( and tables )         |
         |                               |
         |                               |
   PB -> |-------------------------------|
         |///////////////////////////////|
         |////////// undefined //////////|
         |///////////////////////////////|
    0 -> |-------------------------------|
```

Figure 2. Memory layout showing typical register positions during execution of an EM program.

The base registers for the various memory pieces can be stored in target machine registers or memory.

PB	:	program base	points to the base of the instruction address space.
EB	:	external base	points to the base of the data address space.
HB	:	heap base	points to the base of the heap area.
ML	:	memory limit	marks the high end of the addressable data space.

The stack grows from high EM addresses to low EM addresses, and the heap the other way. The memory between SP and HP is not accessible, but may be allocated later to the stack or the heap if needed. The local data area is allocated starting at the high end of memory.

Because EM address 0 is not mapped onto target address 0, a problem arises when pointers are used. If a program pushed a constant, say 6, onto the stack, and then tried to indirect through it, the wrong word would be fetched, because EM address 6 is mapped onto target address EB+6 and not target address 6 itself. This particular problem is solved by explicitly declaring the format of a pointer to be undefined, so that using a constant as a pointer is completely illegal. However, the general problem of mapping pointers still exists.

There are two possible solutions. In the first solution, EM pointers are represented in the target machine as true EM addresses, for example, a pointer to EM address 6 really is stored as a 6 in the target machine. This solution implies that every time a pointer is fetched EB must be added before referencing the target machine's memory. If the target machine has powerful indexing facilities, EB can be kept in a target machine register, and the relocation can indeed be done on every reference to the data address space at a modest cost in speed.

The other solution consists of having EM pointers refer to the true target machine address. Thus the instruction LAE 6 (Load Address of External 6) would push the value of EB+6 onto the stack. When this approach is chosen, back ends must know how to offset from EB, to translate all instructions that manipulate EM addresses. However, the problem is not completely solved, because a front end may have to initialize a pointer in CON or ROM data to point to a global address. This pointer must also be relocated by the back end or the interpreter.

Although the EM stack grows from high to low EM addresses, some machines have hardware PUSH and POP instructions that require the stack to grow upwards. If reasons of efficiency demand the use of these instructions, then EM can be implemented with the memory layout upside down, as shown in figure 3. This is possible because the pointer format is explicitly undefined. The first element of a word array will have a lower physical address than the second element.

```
      |                 |                |                 |
      |      EB=60      |                |        ^        |
      |                 |                |        |        |
      |-----------------|                |-----------------|
  105 |   45   |   44   | 104        214 |   41   |   40   | 215
      |-----------------|                |-----------------|
  103 |   43   |   42   | 102        212 |   43   |   42   | 213
      |-----------------|                |-----------------|
  101 |   41   |   40   | 100        210 |   45   |   44   | 211
      |-----------------|                |-----------------|
      |        |        |                |                 |
      |        v        |                |      EB=255     |
      |                 |                |                 |

           Type A                             Type B
```

Figure 3. Two possible memory implementations.
Numbers within the boxes are EM addresses.
The other numbers are physical addresses.

So, we have two different EM memory implementations:

A − stack downwards

B − stack upwards

For each of these two possibilities we give the translation of the EM instructions to push the third byte of a global data block starting at EM address 40 onto the stack and to load the word at address 40. All translations assume a word and pointer size of two bytes. The target machine used is a PDP-11 augmented with push and pop instructions. Registers 'r0' and 'r1' are used and suffer from sign extension for byte transfers. Push $40 means push the constant 40, not word 40.

The translation of the EM instructions depends on the pointer representation used. For each of the two solutions explained above the translation is given.

First, the translation for the two implementations using EM addresses as pointer representation:

EM		type A		type B	
LAE	40	push	$40	push	$40
ADP	3	pop	r0	pop	r0
		add	$3,r0	add	$3,r0
		push	r0	push	r0
LOI	1	pop	r0	pop	r0
		−		neg	r0
		clr	r1	clr	r1
		bisb	eb(r0),r1	bisb	eb(r0),r1
		push	r1	push	r1
LOE	40	push	eb+40	push	eb-41

The translation for the two implementations, if the target machine address is used as pointer representation, is:

EM		type A		type B	
LAE	40	push	$eb+40	push	$eb-40
ADP	3	pop	r0	pop	r0
		add	$3,r0	sub	$3,r0
		push	r0	push	r0
LOI	1	pop	r0	pop	r0
		clr	r1	clr	r1
		bisb	(r0),r1	bisb	(r0),r1
		push	r1	push	r1
LOE	40	push	eb+40	push	eb-41

The translation presented above is not intended to be optimal. Most machines can handle these simple cases in one or two instructions. It demonstrates, however, the flexibility of the EM design.

There are several possibilities to implement EM on machines with address spaces larger than 64k bytes. For EM with two byte pointers one could allocate instruction and data space each in a separate 64k piece of memory. EM pointers still have to fit in two bytes, but the base registers PB and EB may be loaded in hardware registers wider than 16 bits, if available. EM implementations can also make efficient use of a machine with separate instruction and data space.

EM with 32 bit pointers allows one to make use of machines with large address spaces. In a virtual, segmented memory system one could use a separate segment for each fragment.

6. TYPE REPRESENTATIONS

The representations used for typed objects are not precisely specified by EM. Sometimes we only specify that a typed object occupies a certain amount of space and state no further restrictions. If one wants to have a different representation of the value of an object on the stack one has to use a convert instruction in most cases. We do specify some relations between the representations of types. This allows some intermixed use of operators for different types on the same object(s). For example, the instruction ZER pushes signed and unsigned integers with the value zero and empty sets. ZER has as only argument the size of the object.

The representation of floating point numbers is a good example, it allows widely varying implementations. The only ways to create floating point numbers are via initialization and via conversions from integer numbers. Only by using conversions to integers and comparing two floating point numbers with each other, can these numbers be converted to human readable output. Implementations may use base 10, base 2 or any other base for exponents, and have freedom in choosing the range of exponent and mantissa.

Other types are more precisely described. In the following paragraphs a description will be given of the restrictions imposed on the representation of the types used. A number **n** used in these paragraphs indicates the size of the object in *bits*.

6.1 Unsigned integers

The range of unsigned integers is 0.. $2^{\mathbf{n}}$-1. A binary representation is assumed. The order of the bits within an object is knowingly left unspecified. Discussing bit order within each 8-bit byte is academic, so the only real freedom of this specification lies in the byte order. We really do not care whether an implementation of a 4-byte integer has its bytes in a particular order of significance. This of course means that some sequences of instructions have unpredictable effects. For example:

```
LOC 258 ; STL 0 ; LAL 0 ; LOI 1      ( wordsize >=2 )
```

The value on the stack after executing this sequence can be anything, but will most likely be 1 or 2.

Conversion between unsigned integers of different sizes have to be done with explicit convert instructions. One cannot simply pad an unsigned integer with zero's at either end and expect a correct result.

We assume existence of at least single word unsigned arithmetic in any implementation.

6.2 Signed Integers

The range of signed integers is $-2^{\mathbf{n}-1}$ .. $2^{\mathbf{n}-1}-1$, in other words the range of signed integers of **n** bits using two's complement arithmetic. The representation is the same as for unsigned integers except the range $2^{\mathbf{n}-1}$ .. $2^{\mathbf{n}}-1$ is mapped on the range $-2^{\mathbf{n}-1}$ .. -1. In other words, the most significant bit is used as sign bit. The convert instructions between signed and unsigned integers of the same size can be used to catch errors.

The value $-2^{\mathbf{n}-1}$ is used for undefined signed integers. EM implementations should trap when this value is used in an operation on signed integers. The instruction mask, accessed with SIM and LIM − see chapter 9 − , can be used to disable such traps.

We assume existence of at least single word signed arithmetic in any implementation.

6.3 Floating point values

Floating point values must have a signed mantissa and a signed exponent. Although no base is specified, base 2 is the normal choice, because the FEF instruction pushes the exponent in base 2.

The implementation of floating point arithmetic is optional. The compilers currently in use have runtime parameters for the size of the floating point values they should use. Common choices are 4 and/or 8 bytes.

6.4 Pointers

EM has two kinds of pointers: for instruction and for data space. Each kind can only be used for its own space, conversion between these two subtypes is impossible. We assume that pointers have a range from 0 upwards. Any implementation may have holes in the pointer range between fragments. One can of course not expect to be able to address two megabyte of memory using a 2-byte pointer. Normally, a 2-byte pointer allows up to 65536 bytes of addressable memory.

Pointer representation has one restriction. The pointer with the same representation as the integer zero of the same size should be invalid. Some languages and/or runtime systems represent the nil pointer as zero.

6.5 Bit sets

All bit sets of size **n** are subsets of the set { i | i>=0, i<**n** }. A bit set contains a bit for each element showing its presence or absence. Bit sets are subdivided into words. The word with the lowest EM address governs the subset { i | i>=0, i<**m** }, where **m** is the number of bits in a word. The next higher words each govern the next higher **m** set elements. The relation between a set with size of a word and an unsigned integer word is that the value of the unsigned integer is the summation of the 2^i where i is in the set.

Example: a 2-word bit set (wordsize 2) containing the elements 1, 6, 8, 15, 18, 21, 27 and 28 is composed of two integers, e.g. at addresses 40 and 42. The word at 40 contains the value 33090 (or −32446), the word at 42 contains the value 6180.

7. DESCRIPTORS

Several instructions use descriptors, notably the range check instruction, the array instructions, the goto instruction and the case jump instructions. Descriptors reside in data space. They may be constructed at run time, but more often they are fixed and allocated in ROM data.

All instructions using descriptors, except GTO, have as argument the size of the integers in the descriptor. All implementations have to allow integers of the size of a word in descriptors. All integers popped from the stack and used for indexing or comparing must have the same size as the integers in the descriptor.

7.1 Range check descriptors

Range check descriptors consist of two integers:

1.	lower bound	signed
2.	upper bound	signed

The range check instruction checks an integer on the stack against these bounds and causes a trap if the value is outside the interval. The value itself is neither changed nor removed from the stack.

7.2 Array descriptors

Each array descriptor describes a single dimension. For multi-dimensional arrays, several array instructions are needed to access a single element. Array descriptors contain the following three integers:

1.	lower bound	signed
2.	upper bound − lower bound	unsigned
3.	number of bytes per element	unsigned

The array instructions LAR, SAR and AAR have the pointer to the start of the descriptor as operand on the stack.

The element A[I] is fetched as follows:

1. Stack the address of A (e.g., using LAE or LAL)
2. Stack the value of I (n-byte integer)
3. Stack the pointer to the descriptor (e.g., using LAE)
4. LAR n (n is the size of the integers in the descriptor and I)

All array instructions first pop the address of the descriptor and the index. If the index is not within the bounds specified, a trap occurs. If ok, (I − lower bound) is multiplied by the number of bytes per element (the third word). The result is added to the address of A and replaces A on the stack.

At this point LAR, SAR and AAR diverge. AAR is finished. LAR pops the address and fetches the data item, the size being specified by the descriptor. The usual restrictions for memory access must be obeyed. SAR pops the address and stores the data item now exposed.

7.3 Non-local goto descriptors

The GTO instruction provides a way of returning directly to any active procedure invocation. The argument of the instruction is the address of a descriptor containing three pointers:

1. value of PC after the jump

2. value of SP after the jump

3. value of LB after the jump

GTO replaces the loads PC, SP and LB from the descriptor, thereby jumping to a procedure and removing zero or more frames from the stack. The LB, SP and PC in the descriptor must belong to a dynamically enclosing procedure, because some EM implementations will need to backtrack through the dynamic chain and use the implementation dependent data in frames to restore registers etc.

7.4 Case descriptors

The case jump instructions CSA and CSB both provide multiway branches selected by a case index. Both fetch two operands from the stack: first a pointer to the low address of the case descriptor and then the case index. CSA uses the case index as index in the descriptor table, but CSB searches the table for an occurrence of the case index. Therefore, the descriptors for CSA and CSB, as shown in figure 4, are different. All pointers in the table must be addresses of instructions in the procedure executing the case instruction.

CSA selects the new PC by indexing. If the index, a signed integer, is greater than or equal to the lower bound and less than or equal to the upper bound, then fetch the new PC from the list of instruction pointers by indexing with index-lower. The table does not contain the value of the upper bound, but the value of upper-lower as an unsigned integer. The default instruction pointer is used when the index is out of bounds. If the resulting PC is 0, then trap.

CSB selects the new PC by searching. The table is searched for an entry with index value equal to the case index. That entry or, if none is found, the default entry contains the new PC. When the resulting PC is 0, a trap is performed.

The choice of which case instruction to use for each source language case statement is up to the front end. If the range of the index value is dense, i.e

(highest value − lowest value) / number of cases

is less than some threshold, then CSA is the obvious choice. If the range is sparse, CSB is better.

```
|--------------------|        |--------------------|  high address
| pointer for upb    |        |    pointer n-1     |
|--------------------|        |-  -  -  -  -  -  - |
|         .          |        |     index  n-1     |
|         .          |        |--------------------|
|         .          |        |         .          |
|         .          |        |         .          |
|         .          |        |         .          |
|         .          |        |--------------------|
|         .          |        |    pointer  1      |
|--------------------|        |-  -  -  -  -  -  - |
| pointer for lwb+1  |        |     index   1      |
|--------------------|        |--------------------|
| pointer for lwb    |        |    pointer  0      |
|--------------------|        |-  -  -  -  -  -  - |
|   upper - lower    |        |     index   0      |
|--------------------|        |--------------------|
|    lower bound     |        | number of entries  |
|--------------------|        |--------------------|
|  default pointer   |        |  default pointer   |  low address
|--------------------|        |--------------------|

   CSA descriptor                 CSB descriptor
```

Figure 4. Descriptor layout for CSA and CSB

8. ENVIRONMENT INTERACTIONS

EM programs can interact with their environment in three ways. Two, starting/stopping and monitor calls, are dealt with in this chapter. The remaining way to interact, interrupts, will be treated together with traps in chapter 9.

8.1 Program starting and stopping

EM user programs start with a call to a procedure called _m_a_i_n. The assembler and backends look for the definition of a procedure with this name in their input. The call passes three parameters to the procedure. The parameters are similar to the parameters supplied by the UNIX operating system to C programs. These parameters are often called **argc**, **argv** and **envp**. Argc is the parameter nearest to LB and is a wordsized integer. The other two are pointers to the first element of an array of string pointers.
The **argv** array contains **argc** strings, the first of which contains the program call name. The other strings in the **argv** array are the program parameters.

The **envp** array contains strings in the form "name=string", where ’name’ is the name of an environment variable and string its value. The **envp** is terminated by a zero pointer.

An EM user program stops if the program returns from the first invocation of _m_a_i_n. The contents of the function return area are used to procure a wordsized program return code. EM programs also stop when traps and interrupts occur that are not caught and when the exit monitor call is executed.

8.2 Input/Output and other monitor calls

EM differs from most conventional machines in that it has high level i/o instructions. Typical instructions are OPEN FILE and READ FROM FILE instead of low level instructions such as setting and clearing bits in device registers. By providing such high level i/o primitives, the task of implementing EM on various non EM machines is made considerably easier.

I/O is initiated by the MON instruction, which expects an iocode on top of the stack. Often there are also parameters which are pushed on the stack in reverse order, that is: last parameter first. Some i/o functions also provide results, which are returned on the stack. In the list of monitor calls we use several types of parameters and results, these types consist of integers and unsigneds of varying sizes, but never smaller than the wordsize, and the two pointer types.

The names of the types used are:

int	an integer of wordsize
int2	an integer whose size is the maximum of the wordsize and 2 bytes
int4	an integer whose size is the maximum of the wordsize and 4 bytes
intp	an integer with the size of a pointer
uns2	an unsigned integer whose size is the maximum of the wordsize and 2
unsp	an unsigned integer with the size of a pointer
ptr	a pointer into data space

The table below lists the i/o codes with their results and parameters. This list is similar to the system calls of the UNIX Version 7 operating system.
To execute a monitor call, proceed as follows:

a) Stack the parameters, in reverse order, last parameter first.
b) Push the monitor call number (iocode) onto the stack.

UNIX is a Trademark of Bell Laboratories.

c) Execute the MON instruction.

An error code is present on the top of the stack after execution of most monitor calls. If this error code is zero, the call performed the action requested and the results are available on top of the stack. Non-zero error codes indicate a failure, in this case no results are available and the error code has been pushed twice. This construction enables programs to test for failure with a single instruction (TEQ or TNE) and still find out the cause of the failure. The result name 'e' is reserved for the error code.

List of monitor calls.

nr	name	parameters	results	function
1	Exit	status:int		Terminate this process
2	Fork		e,flag,pid:int	Spawn new process
3	Read	fildes:int;buf:ptr;nbytes:unsp	e:int;rbytes:unsp	Read from file
4	Write	fildes:int;buf:ptr;nbytes:unsp	e:int;wbytes:unsp	Write on a file
5	Open	string:ptr;flag:int	e,fildes:int	Open file for read and/or write
6	Close	fildes:int	e:int	Close a file
7	Wait		e:int;status,pid:int2	Wait for child
8	Creat	string:ptr;mode:int	e,fildes:int	Create a new file
9	Link	string1,string2:ptr	e:int	Link to a file
10	Unlink	string:ptr	e:int	Remove directory entry
12	Chdir	string:ptr	e:int	Change default directory
14	Mknod	string:ptr;mode,addr:int2	e:int	Make a special file
15	Chmod	string:ptr;mode:int2	e:int	Change mode of file
16	Chown	string:ptr;owner,group:int2	e:int	Change owner/group of a file
18	Stat	string,statbuf:ptr	e:int	Get file status
19	Lseek	fildes:int;off:int4;whence:int	e:int;oldoff:int4	Move read/write pointer
20	Getpid		pid:int2	Get process identification
21	Mount	special,string:ptr;rwflag:int	e:int	Mount file system
22	Umount	special:ptr	e:int	Unmount file system
23	Setuid	userid:int2	e:int	Set user ID
24	Getuid		e_uid,r_uid:int2	Get user ID
25	Stime	time:int4	e:int	Set time and date
26	Ptrace	request:int;pid:int2;addr:ptr;data:int	e,value:int	Process trace
27	Alarm	seconds:uns2	previous:uns2	Schedule signal
28	Fstat	fildes:int;statbuf:ptr	e:int	Get file status
29	Pause			Stop until signal
30	Utime	string,timep:ptr	e:int	Set file times
33	Access	string:ptr;mode:int	e:int	Determine file accessibility
34	Nice	incr:int		Set program priority
35	Ftime	bufp:ptr	e:int	Get date and time
36	Sync			Update filesystem

37	Kill	pid:int2;sig:int		
			e:int	Send signal to a process
41	Dup	fildes,newfildes:int		
			e,fildes:int	Duplicate a file descriptor
42	Pipe		e,w_des,r_des:int	Create a pipe
43	Times	buffer:ptr		Get process times
44	Profil	buff:ptr;bufsiz,offset,scale:intp		
				Execution time profile
46	Setgid	gid:int2	e:int	Set group ID
47	Getgid		e_gid,r_gid:int	Get group ID
48	Sigtrp	trapno,signo:int		
			e,prevtrap:int	See below
51	Acct	file:ptr	e:int	Turn accounting on or off
53	Lock	flag:int	e:int	Lock a process
54	Ioctl	fildes,request:int;argp:ptr		
			e:int	Control device
56	Mpxcall	cmd:int;vec:ptr	e:int	Multiplexed file handling
59	Exece	name,argv,envp:ptr		
			e:int	Execute a file
60	Umask	mask:int2	oldmask:int2	Set file creation mode mask
61	Chroot	string:ptr	e:int	Change root directory

Codes 0, 11, 13, 17, 31, 32, 38, 39, 40, 45, 49, 50, 52, 55, 57, 58, 62, and 63 are not used.

All monitor calls, except fork and sigtrp are the same as the UNIX version 7 system calls.

The sigtrp entry maps UNIX signals onto EM interrupts. Normally, trapno is in the range 0 to 252. In that case it requests that signal signo will cause trap trapno to occur. When given trap number −2, default signal handling is reset, and when given trap number −3, the signal is ignored.

The flag returned by fork is 1 in the child process and 0 in the parent. The pid returned is the process-id of the other process.

9. TRAPS AND INTERRUPTS

EM provides a means for the user program to catch all traps generated by the program itself, the hardware, or external conditions. This mechanism uses five instructions: LIM, SIM, SIG, TRP and RTT. This section of the manual may be omitted on the first reading since it presupposes knowledge of the EM instruction set.

The action taken when a trap occurs is determined by the value of an internal EM trap register. This register contains a pointer to a procedure. Initially the pointer used is zero and all traps halt the program with, hopefully, a useful message to the outside world. The SIG instruction can be used to alter the trap register, it pops a procedure pointer from the stack into the trap register. When a trap occurs after storing a nonzero value in the trap register, the procedure pointed to by the trap register is called with the trap number as the only parameter (see below). SIG returns the previous value of the trap register on the stack. Two consecutive SIGs are a no-op. When a trap occurs, the trap register is reset to its initial condition, to prevent recursive traps from hanging the machine up, e.g. stack overflow in the stack overflow handling procedure.

The runtime systems for some languages need to ignore some EM traps. EM offers a feature called the ignore mask. It contains one bit for each of the lowest 16 trap numbers. The bits are numbered 0 to 15, with the least significant bit having number 0. If a certain bit is 1 the corresponding trap never occurs and processing simply continues. The actions performed by the offending instruction are described by the Pascal program in appendix A.
If the bit is 0, traps are not ignored. The instructions LIM and SIM allow copying and replacement of the ignore mask.

The TRP instruction generates a trap, the trap number being found on the stack. This is, among other things, useful for library procedures and runtime systems. It can also be used by a low level trap procedure to pass the trap to a higher level one (see example below).

The RTT instruction returns from the trap procedure and continues after the trap. In the list below all traps marked with an asterisk ('*') are considered to be fatal and it is explicitly undefined what happens when restarting after the trap.

The way a trap procedure is called is completely compatible with normal calling conventions. The only way a trap procedure differs from normal procedures is the return. It has to use RTT instead of RET. This is necessary because the complete runtime status is saved on the stack before calling the procedure and all this status has to be reloaded. Error numbers are in the range 0 to 252. The trap numbers are divided into three categories:

0− 63	EM machine errors, e.g. illegal instruction.	
	0−15	maskable
	16−63	not maskable
64−127	Reserved for use by compilers, run time systems, etc.	
128−252	Available for user programs.	

EM machine errors are numbered as follows:

0	EARRAY	Array bound error
1	ERANGE	Range bound error
2	ESET	Set bound error
3	EIOVFL	Integer overflow
4	EFOVFL	Floating overflow
5	EFUNFL	Floating underflow
6	EIDIVZ	Divide by 0
7	EFDIVZ	Divide by 0.0
8	EIUND	Undefined integer
9	EFUND	Undefined float
10	ECONV	Conversion error
16*	ESTACK	Stack overflow
17	EHEAP	Heap overflow
18*	EILLINS	Illegal instruction
19*	EODDZ	Illegal size argument

20*	ECASE	Case error
21*	EMEMFLT	Addressing non existent memory
22*	EBADPTR	Bad pointer used
23*	EBADPC	Program counter out of range
24	EBADLAE	Bad argument of LAE
25	EBADMON	Bad monitor call
26	EBADLIN	Argument of LIN too high
27	EBADGTO	GTO descriptor error

As an example, suppose a subprocedure has to be written to do a numeric calculation. When an overflow occurs the computation has to be stopped and the higher level procedure must be resumed. This can be programmed as follows using the mechanism described above:

```
 mes 2,2,2               ; set sizes
ersave
 bss 2,0,0               ; Room to save previous value of trap procedure
msave
 bss 2,0,0               ; Room to save previous value of trap mask

 pro $calcule,0          ; entry point
 lxl 0                   ; fill in non-local goto descriptor with LB
 ste jmpbuf+4
 lor 1                   ; and SP
 ste jmpbuf+2
 lim                     ; get current ignore mask
 ste msave               ; save it
 lim
 loc 16                  ; bit for EFOVFL
 ior 2                   ; set in mask
 sim                     ; ignore EFOVFL from now on
 lpi $catch              ; load procedure identifier
 sig                     ; catch wil get all traps now
 ste ersave              ; save previous trap procedure identifier
                         ; perform calculation now, possibly generating overflow
1                        ; label jumped to by catch procedure
 loe ersave              ; get old trap procedure
 sig                     ; refer all following trap to old procedure
 asp 2                   ; remove result of sig
 loe msave               ; restore previous mask
 sim                     ; done now
                         ; load result of calculation
 ret 2                   ; return result
jmpbuf
 con *1,0,0
 end
```

Example of catch procedure

```
 pro $catch,0        ; Local procedure that must catch the overflow trap
 lol 2               ; Load trap number
 loc 4               ; check for overflow
 bne *1              ; if other trap, call higher trap procedure
 gto jmpbuf          ; return to procedure calcule
1                    ; other trap has occurred
 loe ersave          ; previous trap procedure
 sig                 ; other procedure will get the traps now
 asp 2               ; remove the result of sig
 lol 2               ; stack trap number
 trp                 ; call other trap procedure
 rtt                 ; if other procedure returns, do the same
 end
```

10. EM MACHINE LANGUAGE

The EM machine language is designed to make program text compact and to make decoding easy. Compact program text has many advantages: programs execute faster, programs occupy less primary and secondary storage and loading programs into satellite processors is faster. The decoding of EM machine language is so simple, that it is feasible to use interpreters as long as EM hardware machines are not available. This chapter is irrelevant when back ends are used to produce executable target machine code.

10.1 Instruction encoding

A design goal of EM is to make the program text as compact as possible. Decoding must be easy, however. The encoding is fully byte oriented, without any small bit fields. There are 256 primary opcodes, two of which are an escape to two groups of 256 secondary opcodes each.

EM instructions without arguments have a single opcode assigned, possibly escaped:

```
|--------------|
|    opcode    |
|--------------|
```

or

```
|--------------|--------------|
|    escape    |    opcode    |
|--------------|--------------|
```

The encoding for instructions with an argument is more complex. Several instructions have an address from the global data area as argument. Other instructions have different opcodes for positive and negative arguments.

There is always an opcode that takes the next two bytes as argument, high byte first:

```
|--------------|--------------|--------------|
|    opcode    |    hibyte    |    lobyte    |
|--------------|--------------|--------------|
```

or

```
|--------------|--------------|--------------|--------------|
|    escape    |    opcode    |    hibyte    |    lobyte    |
|--------------|--------------|--------------|--------------|
```

An extra escape is provided for instructions with four or eight byte arguments.

```
|--------------|--------------|--------------|   |--------------|
|    ESCAPE    |    opcode    |    hibyte    |...|    lobyte    |
|--------------|--------------|--------------|   |--------------|
```

For most instructions some argument values predominate. The most frequent combinations of instruction and argument will be encoded in a single byte, called a mini:

```
|---------------|
|opcode+argument|  (mini)
|---------------|
```

The number of minis is restricted, because only 254 primary opcodes are available. Many instructions have the bulk of their arguments fall in the range 0 to 255. Instructions that address global data have their arguments distributed over a wider range, but small values of the high byte are common. For all these cases there is another encoding that combines the instruction and the high byte of the argument into a single opcode. These opcodes are called shorties. Shorties may be escaped.

```
|--------------|--------------|
| opcode+high  |    lobyte    |  (shortie)
|--------------|--------------|

              or

|--------------|--------------|--------------|
|    escape    | opcode+high  |    lobyte    |
|--------------|--------------|--------------|
```

Escaped shorties are useless if the normal encoding has a primary opcode. Note that for some instruction-argument combinations several different encodings are available. It is the task of the assembler to select the shortest of these. The savings by these mini and shortie opcodes are considerable, about 55%.

Further improvements are possible: the arguments of many instructions are a multiple of the wordsize. Some do also not allow zero as an argument. If these arguments are divided by the wordsize and, when zero is not allowed, then decremented by 1, more of them can be encoded as shortie or mini. The arguments of some other instructions rarely or never assume the value 0, but start at 1. The value 1 is then encoded as 0, 2 as 1 and so on.

Assigning opcodes to instructions by the assembler is completely table driven. For details see appendix B.

10.2 Procedure descriptors

The procedure identifiers used in the interpreter are indices into a table of procedure descriptors. Each descriptor contains:

1. the number of bytes to be reserved for locals at each invocation.
   This is a pointer-sized integer.

2. the start address of the procedure

10.3 Load format

The EM machine language load format defines the interface between the EM assembler/loader and the EM machine itself. A load file consists of a header, the program text to be executed, a description of the global data area and the procedure descriptor table, in this order. All integers in the load file are presented with the least significant byte first.

The header has two parts: the first half (eight 16-bit integers) aids in selecting the correct EM machine or interpreter. Some EM machines, for instance, may have hardware floating point instructions.
The header entries are as follows (bit 0 is rightmost):

1: magic number (07255)
2: flag bits with the following meaning:
   bit 0 TEST; test for integer overflow etc.
   bit 1 PROFILE; for each source line: count the number of memory cycles executed.
   bit 2 FLOW; for each source line: set a bit in a bit map table if instructions on that line are executed.
   bit 3 COUNT; for each source line: increment a counter if that line is entered.
   bit 4 REALS; set if a program uses floating point instructions.
   bit 5 EXTRA; more tests during compiler debugging.
3: number of unresolved references.
4: version number; used to detect obsolete EM load files.
5: wordsize ; the number of bytes in each machine word.
6: pointer size ; the number of bytes available for addressing.

7: unused
8: unused

The second part of the header (eight entries, of pointer size bytes each) describes the load file itself:

1: NTEXT; the program text size in bytes.
2: NDATA; the number of load-file descriptors (see below).
3: NPROC; the number of entries in the procedure descriptor table.
4: ENTRY; procedure number of the procedure to start with.
5: NLINE; the maximum source line number.
6: SZDATA; the address of the lowest uninitialized data byte.
7: unused
8: unused

The program text consists of NTEXT bytes. NTEXT is always a multiple of the wordsize. The first byte of the program text is the first byte of the instruction address space, i.e. it has address 0. Pointers into the program text are found in the procedure descriptor table where relocation is simple and in the global data area. The initialization of the global data area allows easy relocation of pointers into both address spaces.

The global data area is described by the NDATA descriptors. Each descriptor describes a number of consecutive words (of wordsize) and consists of a sequence of bytes. While reading the descriptors from the load file, one can initialize the global data area from low to high addresses. The size of the initialized data area is given by SZDATA, this number can be used to check the initialization.
The header of each descriptor consists of a byte, describing the type, and a count. The number of bytes used for this (unsigned) count depends on the type of the descriptor and is either a pointer-sized integer or one byte. The meaning of the count depends on the descriptor type. At load time an interpreter can perform any conversion deemed necessary, such as reordering bytes in integers and pointers and adding base addresses to pointers.

In the following pictures we show a graphical notation of the initializers. The leftmost rectangle represents the leading byte.

Fields marked with

n contain a pointer-sized integer used as a count
m contain a one-byte integer used as a count
b contain a one-byte integer
w contain a wordsized integer
p contain a data or instruction pointer
s contain a null terminated ASCII string

```
-------------------
| 0 |      n      |           repeat last initialization n times
-------------------


---------
| 1 | m |                     m uninitialized words
---------


           ____________
          /    bytes   \
-----------------   -----
| 2 | m | b | b |...| b |     m initialized bytes
-----------------   -----
```

```
          _________
         /  word   \
-----------------------
| 3 | m |     w     |...    m initialized wordsized integers
-----------------------


          _________
         / pointer \
-----------------------
| 4 | m |     p     |...    m initialized data pointers
-----------------------


          _________
         / pointer \
-----------------------
| 5 | m |     p     |...    m initialized instruction pointers
-----------------------


          ____________
         /    bytes   \
-------------------------
| 6 | m | b | b |...| b |     initialized integer of size m
-------------------------


          ____________
         /    bytes   \
-------------------------
| 7 | m | b | b |...| b |     initialized unsigned of size m
-------------------------


          ____________
         /   string   \
-------------------------
| 8 | m |      s        |     initialized float of size m
-------------------------
```

type 0: If the last initialization initialized k bytes starting at address *a*, do the same initialization again n times, starting at *a*+k, *a*+2*k, ... *a*+n*k. This is the only descriptor whose starting byte is followed by an integer with the size of a pointer, in all other descriptors the first byte is followed by a one-byte count. This descriptor must be preceded by a descriptor of another type.

type 1: Reserve m words, not explicitly initialized (BSS and HOL).

type 2: The m bytes following the descriptor header are initializers for the next m bytes of the global data area. m is divisible by the wordsize.

type 3: The m words following the header are initializers for the next m words of the global data area.

type 4: The m data address space pointers following the header are initializers for the next m data pointers in the global data area. Interpreters that represent EM pointers by target machine addresses must relocate all data pointers.

type 5: The m instruction address space pointers following the header are initializers for the next m instruction pointers in the global data area. Interpreters that represent EM instruction pointers by target machine addresses must relocate these pointers.

type 6: The m bytes following the header form a signed integer number with a size of m bytes, which is an initializer for the next m bytes of the global data area. m is governed by the same restrictions as for transfer of objects to/from memory.

type 7: The m bytes following the header form an unsigned integer number with a size of m bytes, which is an initializer for the next m bytes of the global data area. m is governed by the same restrictions as for transfer of objects to/from memory.

type 8: The header is followed by an ASCII string, null terminated, to initialize, in global data, a floating point number with a size of m bytes. m is governed by the same restrictions as for transfer of objects to/from memory. The ASCII string contains the notation of a real as used in the Pascal language.

The NPROC procedure descriptors on the load file consist of an instruction space address (of pointer size) and an integer (of pointer size) specifying the number of bytes for locals.

11. EM ASSEMBLY LANGUAGE

We use two representations for assembly language programs, one is in ASCII and the other is the compact assembly language. The latter needs less space than the first for the same program and therefore allows faster processing. Our only program accepting ASCII assembly language converts it to the compact form. All other programs expect compact assembly input. The first part of the chapter describes the ASCII assembly language and its semantics. The second part describes the syntax of the compact assembly language. The last part lists the EM instructions with the type of arguments allowed and an indication of the function. Appendix A gives a detailed description of the effect of all instructions in the form of a Pascal program.

11.1 ASCII assembly language

An assembly language program consists of a series of lines, each line may be blank, contain one (pseudo)instruction or contain one label. Input to the assembler is in lower case. Upper case is used in this document merely to distinguish keywords from the surrounding prose. Comment is allowed at the end of each line and starts with a semicolon ";". This kind of comment does not exist in the compact form.

Labels must be placed all by themselves on a line and start in column 1. There are two kinds of labels, instruction and data labels. Instruction labels are unsigned positive integers. The scope of an instruction label is its procedure.

The pseudoinstructions CON, ROM and BSS may be preceded by a line containing a 1−8 character data label, the first character of which is a letter, period or underscore. The period may only be followed by digits, the others may be followed by letters, digits and underscores. The use of the character "." followed by a constant, which must be in the range 1 to 32767 (e.g. ".40") is recommended for compiler generated programs. These labels are considered as a special case and handled more efficiently in compact assembly language (see below). Note that a data label on its own or two consecutive labels are not allowed.

Each statement may contain an instruction mnemonic or pseudoinstruction. These must begin in column 2 or later (not column 1) and must be followed by a space, tab, semicolon or LF. Everything on the line following a semicolon is taken as a comment.

Each input file contains one module. A module may contain many procedures, which may be nested. A procedure consists of a PRO statement, a (possibly empty) collection of instructions and pseudoinstructions and finally an END statement. Pseudoinstructions are also allowed between procedures. They do not belong to a specific procedure.

All constants in EM are interpreted in the decimal base. The ASCII assembly language accepts constant expressions wherever constants are allowed. The operators recognized are: +, −, *, % and / with the usual precedence order. Use of the parentheses (and) to alter the precedence order is allowed.

11.1.1 Instruction arguments

Unlike many other assembly languages, the EM assembly language requires all arguments of normal and pseudoinstructions to be either a constant or an identifier, but not a combination of these two. There is one exception to this rule: when a data label is used for initialization or as an instruction argument, expressions of the form ’label+constant’ and ’label-constant’ are allowed. This makes it possible to address, for example, the third word of a ten word BSS block directly. Thus LOE LABEL+4 is permitted and so is CON LABEL+3. The resulting address is must be in the same fragment as the label. It is not allowed to add or subtract from instruction labels or procedure identifiers, which certainly is not a severe restriction and greatly aids optimization.

Instruction arguments can be constants, data labels, data labels offsetted by a constant, instruction labels and procedure identifiers. The range of integers allowed depends on the instruction. Most instructions allow only integers (signed or unsigned) that fit in a word. Arguments used as offsets to pointers should fit in a pointer-sized integer. Finally, arguments to LDC should fit in a double-word integer.

Several instructions have two possible forms: with an explicit argument and with an implicit argument on top of the stack. The size of the implicit argument is the wordsize. The implicit argument is always popped before all other operands. For example: ’CMI 4’ specifies that two four-byte signed integers on top of the stack are to be compared. ’CMI’ without an argument expects a wordsized integer on top of the stack that specifies the size of the integers to be compared. Thus the following two sequences are equivalent:

```
LDL   -10
LDL   -14

CMI     4
ZEQ    *1
```

```
LDL   -10
LDL   -14
LOC     4
CMI
ZEQ    *1
```

Section 11.1.6 shows the arguments allowed for each instruction.

11.1.2 Pseudoinstruction arguments

Pseudoinstruction arguments can be divided in two classes: Initializers and others. The following initializers are allowed: signed integer constants, unsigned integer constants, floating-point constants, strings, data labels, data labels offsetted by a constant, instruction labels and procedure identifiers.

Constant initializers in BSS, HOL, CON and ROM pseudoinstructions can be followed by a letter I, U or F. This indicator specifies the type of the initializer: Integer, Unsigned or Float. If no indicator is present I is assumed. The size of the initializer is the wordsize unless the indicator is followed by an integer specifying the initializer's size. This integer is governed by the same restrictions as for transfer of objects to/from memory. As in instruction arguments, initializers include expressions of the form: "LABEL+offset" and "LABEL−offset". The offset must be an unsigned decimal constant. The 'IUF' indicators cannot be used in the offsets.

Data labels are referred to by their name.

Strings are surrounded by double quotes ("). Semicolon's in string do not indicate the start of comment. In the ASCII representation the escape character \ (backslash) alters the meaning of subsequent character(s). This feature allows inclusion of zeroes, graphic characters and the double quote in the string. The following escape sequences exist:

newline	NL (LF)	\n
horizontal tab	HT	\t
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
backslash	\	\\
double quote	"	\"
bit pattern	**ddd**	**ddd**

The escape **ddd** consists of the backslash followed by 1, 2, or 3 octal digits specifying the value of the desired character. If the character following a backslash is not one of those specified, the backslash is ignored. Example: CON "hello\012\0". Each string element initializes a single byte. The ASCII character set is used to map characters onto values.

Instruction labels are referred to as *1, *2, etc. in both branch instructions and as initializers.

The notation $procname means the identifier for the procedure with the specified name. This identifier has the size of a pointer.

11.1.3 Notation

First, the notation used for the arguments, classes of instructions and pseudoinstructions.

<cst>	=	integer constant (current range −2**31..2**31−1)
<dlb>	=	data label
<arg>	=	<cst> or <dlb> or <dlb>+<cst> or <dlb>−<cst>
<con>	=	integer constant, unsigned constant, floating-point constant
<str>	=	string constant (surrounded by double quotes),
<ilb>	=	instruction label '*' followed by an integer in the range 0..32767.
<pro>	=	procedure number ('$' followed by a procedure name)
<val>	=	<arg>, <con>, <pro> or <ilb>.

<par> = <val> or <str>
<...>* = zero or more of <...>
<...>+ = one or more of <...>
[...] = optional ...

11.1.4 Pseudoinstructions

11.1.4.1 Storage declaration

Initialized global data is allocated by the pseudoinstruction CON, which needs at least one argument. Each argument is used to allocate and initialize a number of consecutive bytes in data memory. The number of bytes to be allocated and the alignment depend on the type of the argument. For each argument, an integral number of words, determined by the argument type, is allocated and initialized.

The pseudoinstruction ROM is the same as CON, except that it guarantees that the initialized words will not change during the execution of the program. This information allows optimizers to do certain calculations such as array indexing and subrange checking at compile time instead of at run time.

The pseudoinstruction BSS allocates uninitialized global data or large blocks of data initialized by the same value. The first argument to this pseudo is the number of bytes required, which must be a multiple of the wordsize. The other arguments specify the value used for initialization and whether the initialization is only for convenience or a strict necessity. The pseudoinstruction HOL is similar to BSS in that it requests an (un)initialized global data block. Addressing of a HOL block, however, is quasi absolute. The first byte is addressed by 0, the second byte by 1 etc. in assembly language. The assembler/loader adds the base address of the HOL block to these numbers to obtain the absolute address in the machine language.

The scope of a HOL block starts at the HOL pseudo and ends at the next HOL pseudo or at the end of a module whatever comes first. Each instruction falls in the scope of at most one HOL block, the current HOL block. It is not allowed to have more than one HOL block per procedure.

The alignment restrictions are enforced by the pseudoinstructions. All initializers are aligned on a multiple of their size or the wordsize whichever is smaller. Strings form an exception, they are to be seen as a sequence of initializers each for one byte, i.e. strings are not padded with zero bytes. Switching to another type of fragment or placing a label forces word-alignment. There are three types of fragments in global data space: CON, ROM and BSS/HOL.

BSS <cst1>,<val>,<cst2>
: Reserve <cst1> bytes. <val> is the value used to initialize the area. <cst1> must be a multiple of the size of <val>. <cst2> is 0 if the initialization is not strictly necessary, 1 if it is.

HOL <cst1>,<val>,<cst2>
: Idem, but all following absolute global data references will refer to this block. Only one HOL is allowed per procedure, it has to be placed before the first instruction.

CON <val>+
: Assemble global data words initialized with the <val> constants.

ROM <val>+
: Idem, but the initialized data will never be changed by the program.

11.1.4.2 Partitioning

Two pseudoinstructions partition the input into procedures:

PRO <pro>[,<cst>]
: Start of procedure. <pro> is the procedure name. <cst> is the number of bytes for locals. The number of bytes for locals must be specified in the PRO or END pseudoinstruction. When specified in both, they must be identical.

END [<cst>]

End of Procedure. <cst> is the number of bytes for locals. The number of bytes for locals must be specified in either the PRO or END pseudoinstruction or both.

11.1.4.3 Visibility

Names of data and procedures in an EM module can either be internal or external. External names are known outside the module and are used to link several pieces of a program. Internal names are not known outside the modules they are used in. Other modules will not 'see' an internal name.

To reduce the number of passes needed, it must be known at the first occurrence whether a name is internal or external. If the first occurrence of a name is in a definition, the name is considered to be internal. If the first occurrence of a name is a reference, the name is considered to be external. If the first occurrence is in one of the following pseudoinstructions, the effect of the pseudo has precedence.

EXA <dlb>

External name. <dlb> is known, possibly defined, outside this module. Note that <dlb> may be defined in the same module.

EXP <pro>

External procedure identifier. Note that <pro> may be defined in the same module.

INA <dlb>

Internal name. <dlb> is internal to this module and must be defined in this module.

INP <pro>

Internal procedure. <pro> is internal to this module and must be defined in this module.

11.1.4.4 Miscellaneous

Two other pseudoinstructions provide miscellaneous features:

EXC <cst1>,<cst2>

Two blocks of instructions preceding this one are interchanged before being processed. <cst1> gives the number of lines of the first block. <cst2> gives the number of lines of the second one. Blank and pure comment lines do not count. This instruction is obsolete. Its use is strongly discouraged.

MES <cst>[,<par>]*

A special type of comment. Used by compilers to communicate with the optimizer, assembler, etc. as follows:

MES 0

An error has occurred, stop further processing.

MES 1

Suppress optimization.

MES 2,<cst1>,<cst2>

Use wordsize <cst1> and pointer size <cst2>.

MES 3,<cst1>,<cst2>,<cst3>,<cst4>

Indicates that a local variable is never referenced indirectly. Used to indicate that a register may be used for a specific variable. <cst1> is offset in bytes from AB if positive and offset from LB if negative. <cst2> gives the size of the variable. <cst3> indicates the class of the variable. The following values are currently recognized:

0 The variable can be used for anything.
1 The variable is used as a loopindex.
2 The variable is used as a pointer.
3 The variable is used as a floating point number.

<cst4> gives the priority of the variable, higher numbers indicate better candidates.

MES 4,<cst>,<str>

Number of source lines in file <str> (for profiler).

MES 5
: Floating point used.

MES 6,<val>*
: Comment. Used to provide comments in compact assembly language.

MES 7,.....
: Reserved.

MES 8,<pro>[,<dlb>]...
: Library module. Indicates that the module may only be loaded if it is useful, that is, if it can satisfy any unresolved references during the loading process. May not be preceded by any other pseudo, except MES's.

MES 9,<cst>
: Guarantees that no more than <cst> bytes of parameters are accessed, either directly or indirectly.

MES 10,<cst>[,<par>]*
: This message number is reserved for the global optimizer. It inserts these messages in its output as hints to backends. <cst> indicates the type of hint.

MES 11
: Procedures containing this message are possible destinations of non-local goto's with the GTO instruction. Some backends keep locals in registers, the locals in this procedure should not be kept in registers and all registers containing locals of other procedures should be saved upon entry to this procedure.

Each backend is free to skip irrelevant MES pseudos.

11.2 The Compact Assembly Language

The assembler accepts input in a highly encoded form. This form is intended to reduce the amount of file transport between the front ends, optimizers and back ends, and also reduces the amount of storage required for storing libraries. Libraries are stored as archived compact assembly language, not machine language.

When beginning to read the input, the assembler is in neutral state, and expects either a label or an instruction (including the pseudoinstructions). The meaning of the next byte(s) when in neutral state is as follows, where b1, b2 etc. represent the succeeding bytes.

0	Reserved for future use
1−129	Machine instructions, see Appendix A, alphabetical list
130−149	Reserved for future use
150−161	BSS,CON,END,EXA,EXC,EXP,HOL,INA,INP,MES,PRO,ROM
162−179	Reserved for future pseudoinstructions
180−239	Instruction labels 0 − 59 (180 is local label 0 etc.)
240−244	See the Common Table below
245−255	Not used

After a label, the assembler is back in neutral state; it can immediately accept another label or an instruction in the next byte. No linefeeds are used to separate lines.

If an opcode expects no arguments, the assembler is back in neutral state after reading the one byte containing the instruction number. If it has one or more arguments (only pseudos have more than 1), the arguments follow directly, encoded as follows:

0−239	Offsets from −120 to 119
240−255	See the Common Table below

Absence of an optional argument is indicated by a special byte.

Common Table for Neutral State and Arguments

class		bytes	description
<ilb>	240	b1	Instruction label b1 (Not used for branches)
<ilb>	241	b1 b2	16 bit instruction label (256*b2 + b1)
<dlb>	242	b1	Global label .0−.255, with b1 being the label
<dlb>	243	b1 b2	Global label .0−.32767 with 256*b2+b1 being the label
<dlb>	244	<string>	Global symbol not of the form .nnn
<cst>	245	b1 b2	16 bit constant
<cst>	246	b1 b2 b3 b4	32 bit constant
<cst>	247	b1 .. b8	64 bit constant
<arg>	248	<dlb><cst>	Global label + (possibly negative) constant
<pro>	249	<string>	Procedure name (not including $)
<str>	250	<string>	String used in CON or ROM (no quotes-no escapes)
<con>	251	<cst><string>	Integer constant, size <cst> bytes
<con>	252	<cst><string>	Unsigned constant, size <cst> bytes
<con>	253	<cst><string>	Floating constant, size <cst> bytes
	254		unused
<end>	255		Delimiter for argument lists or indicates absence of optional argument

The bytes specifying the value of a 16, 32 or 64 bit constant are presented in two’s complement notation, with the least significant byte first. For example: the value of a 32 bit constant is ((s4*256+b3)*256+b2)*256+b1, where s4 is b4−256 if b4 is greater than 128 else s4 takes the value of b4. A <string> consists of a <cst> immediately followed by a sequence of bytes with length <cst>.

The pseudoinstructions fall into several categories, depending on their arguments:

Group 1 − EXC, BSS, HOL have a known number of arguments
Group 2 − EXA, EXP, INA, INP have a string as argument
Group 3 − CON, MES, ROM have a variable number of various things
Group 4 − END, PRO have a trailing optional argument.

Groups 1 and 2 use the encoding described above. Group 3 also uses the encoding listed above, with an <end> byte after the last argument to indicate the end of the list. Group 4 uses an <end> byte if the trailing argument is not present.

Example ASCII (LOC = 69, BRA = 18 here):	Example compact
2	182
1	181
LOC 10	69 130
LOC −10	69 110
LOC 300	69 245 44 1
BRA *19	18 139
300	241 44 1
.3	242 3
CON 4,9,*2,$foo	151 124 129 240 2 249 123 102 111 111 255
CON .35	151 242 35 255

11.3 Assembly language instruction list

For each instruction in the list the range of argument values in the assembly language is given. The column headed *assem* contains the mnemonics defined in 11.1.3. The following column specifies restrictions of the argument value. Addresses have to obey the restrictions mentioned in chapter 2. The classes of arguments are indicated by letters:

	assem	constraints	rationale
c	cst	fits word	constant
d	cst	fits double word	constant
l	cst		local offset
g	arg	>= 0	global offset
f	cst		fragment offset
n	cst	>= 0	counter
s	cst	>0 , word multiple	object size
z	cst	>= 0 , zero or word multiple	object size
o	cst	> 0 , word multiple or fraction	object size
w	cst	> 0 , word multiple	object size *
p	pro		pro identifier
b	ilb	>= 0	label number
r	cst	0,1,2	register number
−			no argument

The * at the rationale for **w** indicates that the argument can either be given as argument or on top of the stack. If the argument is omitted, the argument is fetched from the stack; it is assumed to be a wordsized unsigned integer. Instructions that check for undefined integer or floating-point values and underflow or overflow are indicated below by (*).

GROUP 1 − LOAD

LOC **c** :	Load constant (i.e. push one word onto the stack)
LDC **d** :	Load double constant (push two words)
LOL **l** :	Load word at **l**-th local (**l**<0) or parameter (**l**>=0)
LOE **g** :	Load external word **g**
LIL **l** :	Load word pointed to by **l**-th local or parameter
LOF **f** :	Load offsetted (top of stack + **f** yield address)
LAL **l** :	Load address of local or parameter
LAE **g** :	Load address of external
LXL **n** :	Load lexical (address of LB **n** static levels back)
LXA **n** :	Load lexical (address of AB **n** static levels back)
LOI **o** :	Load indirect **o** bytes (address is popped from the stack)
LOS **w** :	Load indirect, **w**-byte integer on top of stack gives object size
LDL **l** :	Load double local or parameter (two consecutive words are stacked)
LDE **g** :	Load double external (two consecutive externals are stacked)
LDF **f** :	Load double offsetted (top of stack + **f** yield address)
LPI **p** :	Load procedure identifier

GROUP 2 − STORE

STL **l** : Store local or parameter
STE **g** : Store external
SIL **l** : Store into word pointed to by **l**-th local or parameter
STF **f** : Store offsetted
STI **o** : Store indirect **o** bytes (pop address, then data)
STS **w** : Store indirect, **w**-byte integer on top of stack gives object size
SDL **l** : Store double local or parameter
SDE **g** : Store double external
SDF **f** : Store double offsetted

GROUP 3 − INTEGER ARITHMETIC

ADI **w** : Addition (*)
SBI **w** : Subtraction (*)
MLI **w** : Multiplication (*)
DVI **w** : Division (*)
RMI **w** : Remainder (*)
NGI **w** : Negate (two's complement) (*)
SLI **w** : Shift left (*)
SRI **w** : Shift right (*)

GROUP 4 − UNSIGNED ARITHMETIC

ADU **w** : Addition
SBU **w** : Subtraction
MLU **w** : Multiplication
DVU **w** : Division
RMU **w** : Remainder
SLU **w** : Shift left
SRU **w** : Shift right

GROUP 5 − FLOATING POINT ARITHMETIC

ADF **w** : Floating add (*)
SBF **w** : Floating subtract (*)
MLF **w** : Floating multiply (*)
DVF **w** : Floating divide (*)
NGF **w** : Floating negate (*)
FIF **w** : Floating multiply and split integer and fraction part (*)
FEF **w** : Split floating number in exponent and fraction part (*)

GROUP 6 − POINTER ARITHMETIC

ADP **f** : Add **f** to pointer on top of stack
ADS **w** : Add **w**-byte value and pointer
SBS **w** : Subtract pointers in same fragment and push diff as size **w** integer

GROUP 7 − INCREMENT/DECREMENT/ZERO

INC − :	Increment word on top of stack by 1 (*)
INL **l** :	Increment local or parameter (*)
INE **g** :	Increment external (*)
DEC − :	Decrement word on top of stack by 1 (*)
DEL **l** :	Decrement local or parameter (*)
DEE **g** :	Decrement external (*)
ZRL **l** :	Zero local or parameter
ZRE **g** :	Zero external
ZRF **w** :	Load a floating zero of size **w**
ZER **w** :	Load **w** zero bytes

GROUP 8 − CONVERT (stack:source, source size, dest. size (top))

CII − :	Convert integer to integer (*)
CUI − :	Convert unsigned to integer (*)
CFI − :	Convert floating to integer (*)
CIF − :	Convert integer to floating (*)
CUF − :	Convert unsigned to floating (*)
CFF − :	Convert floating to floating (*)
CIU − :	Convert integer to unsigned
CUU − :	Convert unsigned to unsigned
CFU − :	Convert floating to unsigned

GROUP 9 − LOGICAL

AND **w** :	Boolean and on two groups of **w** bytes
IOR **w** :	Boolean inclusive or on two groups of **w** bytes
XOR **w** :	Boolean exclusive or on two groups of **w** bytes
COM **w** :	Complement (one’s complement of top **w** bytes)
ROL **w** :	Rotate left a group of **w** bytes
ROR **w** :	Rotate right a group of **w** bytes

GROUP 10 − SETS

INN **w** :	Bit test on **w** byte set (bit number on top of stack)
SET **w** :	Create singleton **w** byte set with bit n on (n is top of stack)

GROUP 11 − ARRAY

LAR **w** :	Load array element, descriptor contains integers of size **w**
SAR **w** :	Store array element
AAR **w** :	Load address of array element

GROUP 12 − COMPARE

CMI **w** :	Compare **w** byte integers, Push negative, zero, positive for <, = or >
CMF **w** :	Compare **w** byte reals
CMU **w** :	Compare **w** byte unsigneds
CMS **w** :	Compare **w** byte values, can only be used for bit for bit equality test
CMP − :	Compare pointers
TLT − :	True if less, i.e. iff top of stack < 0
TLE − :	True if less or equal, i.e. iff top of stack <= 0
TEQ − :	True if equal, i.e. iff top of stack = 0
TNE − :	True if not equal, i.e. iff top of stack non zero
TGE − :	True if greater or equal, i.e. iff top of stack >= 0
TGT − :	True if greater, i.e. iff top of stack > 0

GROUP 13 − BRANCH

BRA **b** :	Branch unconditionally to label **b**
BLT **b** :	Branch less (pop 2 words, branch if top > second)
BLE **b** :	Branch less or equal
BEQ **b** :	Branch equal
BNE **b** :	Branch not equal
BGE **b** :	Branch greater or equal
BGT **b** :	Branch greater
ZLT **b** :	Branch less than zero (pop 1 word, branch negative)
ZLE **b** :	Branch less or equal to zero
ZEQ **b** :	Branch equal zero
ZNE **b** :	Branch not zero
ZGE **b** :	Branch greater or equal zero
ZGT **b** :	Branch greater than zero

GROUP 14 − PROCEDURE CALL

CAI − :	Call procedure (procedure identifier on stack)
CAL **p** :	Call procedure (with identifier **p**)
LFR **s** :	Load function result
RET **z** :	Return (function result consists of top **z** bytes)

GROUP 15 − MISCELLANEOUS

ASP **f** :	Adjust the stack pointer by **f**
ASS **w** :	Adjust the stack pointer by **w**-byte integer
BLM **z** :	Block move **z** bytes; first pop destination addr, then source addr
BLS **w** :	Block move, size is in **w**-byte integer on top of stack
CSA **w** :	Case jump; address of jump table at top of stack
CSB **w** :	Table lookup jump; address of jump table at top of stack
DCH − :	Follow dynamic chain, convert LB to LB of caller
DUP **s** :	Duplicate top **s** bytes
DUS **w** :	Duplicate top **w** bytes
EXG **w** :	Exchange top **w** bytes
FIL **g** :	File name (external 4 := **g**)
GTO **g** :	Non-local goto, descriptor at **g**
LIM − :	Load 16 bit ignore mask
LIN **n** :	Line number (external 0 := **n**)
LNI − :	Line number increment
LOR **r** :	Load register (0=LB, 1=SP, 2=HP)
LPB − :	Convert local base to argument base
MON − :	Monitor call
NOP − :	No operation
RCK **w** :	Range check; trap on error
RTT − :	Return from trap
SIG − :	Trap errors to proc identifier on top of stack, −2 resets default
SIM − :	Store 16 bit ignore mask
STR **r** :	Store register (0=LB, 1=SP, 2=HP)
TRP − :	Cause trap to occur (Error number on stack)

A. EM INTERPRETER

```
 { This  is an interpreter for EM.  It serves as  the official machine
definition.  This interpreter must run on a machine which supports
arithmetic with words and memory offsets.

Certain aspects of the definition are over specified.  In particular:

  1. The representation of  an  address on the stack  need not be the
     numerical value of the memory location.

  2. The state of  the stack is not defined  after a trap has aborted
     an instruction in the middle.  For example, it is officially un-
     defined  whether the second  operand of an  ADD  instruction has
     been popped  or  not  if the  first one is undefined ( -32768 or
     unsigned 32768).

  3. The memory layout is implementation dependent. Only the most
     basic checks are performed whenever memory is accessed.

  4. The representation of an integer or set on the stack is not fixed
     in bit order.

  5. The format and existence of the procedure descriptors depends on
     the implementation.

  6. The result of the compare operators  CMI etc.  are -1, 0  and  1
     here, but other negative  and  positive values will do  and they
     need not be the same each time.

  7. The shift count for SHL, SHR, ROL and ROR must be in the range 0
     to object size in bits - 1.  The effect of a  count  not in this
     range is undefined.
}
```

```
{$i256} {$d+}
program em(tables,prog,input,output);

label 8888,9999;

const
  t15   = 32768;        { 2**15    }
  t15m1 = 32767;        { 2**15 -1 }
  t16   = 65536;        { 2**16    }
  t16m1 = 65535;        { 2**16 -1 }
  t31m1 = 2147483647;   { 2**31 -1 }

  wsize = 2;            { number of bytes in a word }
  asize = 2;            { number of bytes in an address }
  fsize = 4;            { number of bytes in a floating point number }
  maxret =4;            { number of words in the return value area }

  signbit = t15;        { the power of two indicating the sign bit }
  negoff  = t16;        { the next power of two }
  maxsint = t15m1;      { the maximum signed integer }
  maxuint = t16m1;      { the maximum unsigned integer }
  maxdbl  = t31m1;      { the maximum double signed integer }
  maxadr  = t16m1;      { the maximum address }
  maxoffs = t15m1;      { the maximum offset from an address }
  maxbitnr= 15;         { the number of the highest bit }

  lineadr = 0;          { address of the line number }
  fileadr = 4;          { address of the file name }
  maxcode = 8191;       { highest byte in code address space }
  maxdata = 8191;       { highest byte in data address space }

  { format of status save area }
  statd   = 4;          { how far is static link from lb }
  dynd    = 2;          { how far is dynamic link from lb }
  reta    = 0;          { how far is the return address from lb }
  savsize = 4;          { size of save area in bytes }

  { procedure descriptor format }
  pdlocs  = 0;          { offset for size of local variables in bytes }
  pdbase  = asize;      { offset for the procedure base }
  pdsize  = 4;          { size of procedure descriptor in bytes = 2*asize }

  { header words }
  NTEXT   = 1;
  NDATA   = 2;
  NPROC   = 3;
  ENTRY   = 4;
  NLINE   = 5;
  SZDATA  = 6;

  escape1 = 254;        { escape to secondary opcodes }
  escape2 = 255;        { escape to tertiary opcodes }
  undef   = signbit;    { the range of integers is -32767 to +32767 }

  { error codes }
  EARRAY   =  0; ERANGE   =  1; ESET     =  2; EIOVFL   =  3; EFOVFL   =  4;
  EFUNFL   =  5; EIDIVZ   =  6; EFDIVZ   =  7; EIUND    =  8; EFUND    =  9;
  ECONV    = 10; ESTACK   = 16; EHEAP    = 17; EILLINS  = 18; EODDZ    = 19;
  ECASE    = 20; EMEMFLT  = 21; EBADPTR  = 22; EBADPC   = 23; EBADLAE  = 24;
```

```
  EBADMON  = 25; EBADLIN  = 26; EBADGTO  = 27;
```

```
{---------------------------------------------------------------------------}
{                             Declarations                                   }
{---------------------------------------------------------------------------}

type
  bitval= 0..1;             { one bit }
  bitnr=  0..maxbitnr;      { bits in machine words are numbered 0 to 15 }
  byte=   0..255;           { memory is an array of bytes }
  adr=    {0..maxadr} long; { the range of addresses }
  word=   {0..maxuint} long;{ the range of unsigned integers }
  offs=  -maxoffs..maxoffs; { the range of signed offsets from addresses }
  size=   0..maxoffs;       { the range of sizes is the positive offsets }
  sword= {-signbit..maxsint} long; { the range of signed integers }
  full=  {-maxuint..maxuint} long; { intermediate results need this range }
  double={-maxdbl..maxdbl} long;   { double precision range }
  bftype= (andf,iorf,xorf); { tells which boolean operator needed }
  insclass=(prim,second,tert); { tells which opcode table is in use }
  instype=(implic,explic);  { does opcode have implicit or explicit operand }
  iflags= (mini,short,sbit,wbit,zbit,ibit);
  ifset=  set of iflags;

  mnem = ( NON,
           AAR, ADF, ADI, ADP, ADS, ADU,XAND, ASP, ASS, BEQ,
           BGE, BGT, BLE, BLM, BLS, BLT, BNE, BRA, CAI, CAL,
           CFF, CFI, CFU, CIF, CII, CIU, CMF, CMI, CMP, CMS,
           CMU, COM, CSA, CSB, CUF, CUI, CUU, DCH, DEC, DEE,
           DEL, DUP, DUS, DVF, DVI, DVU, EXG, FEF, FIF, FIL,
           GTO, INC, INE, INL, INN, IOR, LAE, LAL, LAR, LDC,
           LDE, LDF, LDL, LFR, LIL, LIM, LIN, LNI, LOC, LOE,
           LOF, LOI, LOL, LOR, LOS, LPB, LPI, LXA, LXL, MLF,
           MLI, MLU, MON, NGF, NGI, NOP, RCK, RET, RMI, RMU,
           ROL, ROR, RTT, SAR, SBF, SBI, SBS, SBU, SDE, SDF,
           SDL,XSET, SIG, SIL, SIM, SLI, SLU, SRI, SRU, STE,
           STF, STI, STL, STR, STS, TEQ, TGE, TGT, TLE, TLT,
           TNE, TRP, XOR, ZEQ, ZER, ZGE, ZGT, ZLE, ZLT, ZNE,
           ZRE, ZRF, ZRL);

  dispatch = record
               iflag: ifset;
               instr: mnem;
               case instype of
               implic: (implicit:sword);
               explic: (ilength:byte);
             end;


var
  code: packed array[0..maxcode] of byte;      { code space }
  data: packed array[0..maxdata] of byte;      { data space }
  retarea: array[1..maxret ] of word;          { return area }
  pc,lb,sp,hp,pd: adr;  { internal machine registers }
  i: integer;           { integer scratch variable }
  s,t :word;            { scratch variables }
  sz:size;              { scratch variables }
  ss,st: sword;         { scratch variables }
  k :double;            { scratch variables }
  j:size;               { scratch variable used as index }
  a,b:adr;              { scratch variable used for addresses }
  dt,ds:double;         { scratch variables for double precision }
```

```
  rt,rs,x,y:real;       { scratch variables for real }
  found:boolean;        { scratch }
  opcode: byte;         { holds the opcode during execution }
  iclass: insclass;     { true for escaped opcodes }
  dispat: array[insclass,byte] of dispatch;
  retsize:size;         { holds size of last LFR }
  insr: mnem;           { holds the instruction number }
  halted: boolean;      { normally false }
  exitstatus:word;      { parameter of MON 1 }
  ignmask:word;         { ignore mask for traps }
  uerrorproc:adr;       { number of user defined error procedure }
  intrap:boolean;       { Set when executing trap(), to catch recursive calls}
  trapval:byte;         { Set to number of last trap }
  header: array[1..8] of adr;

  tables: text;         { description of EM instructions }
  prog: file of byte;   { program and initialized data }


{---------------------------------------------------------------------------}
{                        Various check routines                             }
{---------------------------------------------------------------------------}

{ Only the most basic checks are performed. These routines are inherently
  implementation dependent. }

procedure trap(n:byte); forward;

procedure memadr(a:adr);
begin if (a>maxdata) or ((a<sp) and (a>=hp)) then trap(EMEMFLT) end;

procedure wordadr(a:adr);
begin memadr(a); if (a mod wsize<>0) then trap(EBADPTR) end;

procedure chkadr(a:adr; s:size);
begin memadr(a); memadr(a+s-1); { assumption: size is ok }
      if s<wsize
      then begin if a mod s<>0  then trap(EBADPTR) end
      else       if a mod wsize<>0 then trap(EBADPTR)
end;

procedure newpc(a:double);
begin if (a<0) or (a>maxcode) then trap(EBADPC); pc:=a end;

procedure newsp(a:adr);
begin if (a>lb) or (a<hp) or (a mod wsize<>0) then trap(ESTACK); sp:=a end;

procedure newlb(a:adr);
begin if (a<sp) or (a mod wsize<>0) then trap(ESTACK); lb:=a end;

procedure newhp(a:adr);
begin if (a>sp) or (a>maxdata+1) or (a mod wsize<>0)
      then trap(EHEAP)
      else hp:=a
end;

function argc(a:double):sword;
begin if (a<-signbit) or (a>maxsint) then trap(EILLINS); argc:=a end;
```

```
function argd(a:double):double;
begin if (a<-maxdbl) or (a>maxdbl) then trap(EILLINS); argd:=a end;

function argl(a:double):offs;
begin if (a<-maxoffs) or (a>maxoffs) then trap(EILLINS); argl:=a end;

function argg(k:double):adr;
begin if (k<0) or (k>maxadr) then trap(EILLINS); argg:=k end;

function argf(a:double):offs;
begin if (a<-maxoffs) or (a>maxoffs) then trap(EILLINS); argf:=a end;

function argn(a:double):word;
begin if (a<0) or (a>maxuint) then trap(EILLINS); argn:=a end;

function args(a:double):size;
begin if (a<=0) or (a>maxoffs)
        then trap(EODDZ)
        else if (a mod wsize)<>0 then trap(EODDZ);
      args:=a ;
end;

function argz(a:double):size;
begin if (a<0) or (a>maxoffs)
        then trap(EODDZ)
        else if (a mod wsize)<>0 then trap(EODDZ);
      argz:=a ;
end;

function argo(a:double):size;
begin if (a<=0) or (a>maxoffs)
        then trap(EODDZ)
        else if (a mod wsize<>0) and (wsize mod a<>0) then trap(EODDZ);
      argo:=a ;
end;

function argw(a:double):size;
begin if (a<=0) or (a>maxoffs) or (a>maxuint)
        then trap(EODDZ)
        else if (a mod wsize)<>0 then trap(EODDZ);
      argw:=a ;
end;

function argp(a:double):size;
begin if (a<0) or (a>=header[NPROC]) then trap(EILLINS); argp:=a end;

function argr(a:double):word;
begin if (a<0) or (a>2) then trap(EILLINS); argr:=a end;

procedure argwf(s:double);
begin if argw(s)<>fsize then trap(EILLINS) end;

function szindex(s:double):integer;
begin s:=argw(s); if (s mod wsize <> 0) or (s>2*wsize) then trap(EILLINS);
      szindex:=s div wsize
end;

function locadr(l:double):adr;
begin l:=argl(l); if l<0 then locadr:=lb+l else locadr:=lb+l+savsize end;
```

```
function signwd(w:word):sword;
begin if w = undef then trap(EIUND);
      if w >= signbit then signwd:=w-negoff else signwd:=w
end;

function dosign(w:word):sword;
begin if w >= signbit then dosign:=w-negoff else dosign:=w end;

function unsign(w:sword):word;
begin if w<0 then unsign:=w+negoff else unsign:=w end;

function chopw(dw:double):word;
begin chopw:=dw mod negoff end;

function fitsw(w:full;trapno:byte):word;
{ checks whether value fits in signed word, returns unsigned representation}
begin
  if (w>maxsint) or (w<-signbit) then
    begin trap(trapno);
      if w<0 then fitsw:=negoff- (-w)mod negoff
             else fitsw:=w mod negoff;
    end
  else fitsw:=unsign(w)
end;

function fitd(w:full):double;
begin
  if abs(w) > maxdbl then trap(ECONV);
  fitd:=w
end;


{---------------------------------------------------------------------------}
{                        Memory access routines                             }
{---------------------------------------------------------------------------}

{ memw returns a machine word as an unsigned integer
  memb returns a single byte as a positive integer: 0 <= memb <= 255
  mems(a,s) fetches an object smaller than a word and returns a word
  store(a,v) stores the word v at machine address a
  storea(a,v) stores the address v at machine address a
  storeb(a,b) stores the byte b at machine address a
  stores(a,s,v) stores the s least significant bytes of a word at address a
  memi returns an offset from the instruction space
       Note that the procedure descriptors are part of instruction space.
  nextpc returns the next byte addressed by pc, incrementing pc

  lino changes the line number word.
  filna changes the pointer to the file name.

  All routines check to make sure the address is within range and valid for
  the size of the object. If an addressing error is found, a trap occurs.
}


function memw(a:adr):word;
var b:word; i:integer;
begin wordadr(a); b:=0;
      for i:=wsize-1 downto 0 do b:=256*b + data[a+i] ;
```

```
      memw:=b
end;

function memd(a:adr):double; { Always signed }
var b:double; i:integer;
begin wordadr(a); b:=data[a+2*wsize-1];
      if b>=128 then b:=b-256;
      for i:=2*wsize-2 downto 0 do b:=256*b + data[a+i] ;
      memd:=b
end;

function mema(a:adr):adr;
var b:adr; i:integer;
begin wordadr(a); b:=0;
      for i:=asize-1 downto 0 do b:=256*b + data[a+i] ;
      mema:=b
end;

function mems(a:adr;s:size):word;
var i:integer; b:word;
begin chkadr(a,s); b:=0; for i:=1 to s do b:=b*256+data[a+s-i]; mems:=b end;

function memb(a:adr):byte;
begin memadr(a); memb:=data[a] end;

procedure store(a:adr; x:word);
var i:integer;
begin wordadr(a);
  for i:=0 to wsize-1 do
     begin data[a+i]:=x mod 256; x:=x div 256 end
end;

procedure storea(a:adr; x:adr);
var i:integer;
begin wordadr(a);
  for i:=0 to asize-1 do
     begin data[a+i]:=x mod 256; x:=x div 256 end
end;

procedure stores(a:adr;s:size;v:word);
var i:integer;
begin chkadr(a,s);
  for i:=0 to s-1 do begin data[a+i]:=v mod 256; v:=v div 256 end;
end;

procedure storeb(a:adr; b:byte);
begin memadr(a); data[a]:=b end;

function memi(a:adr):adr;
var b:adr; i:integer;
begin if (a mod wsize<>0) or (a+asize-1>maxcode) then trap(EBADPTR); b:=0;
      for i:=asize-1 downto 0 do b:=256*b + code[a+i] ;
      memi:=b
end;

function nextpc:byte;
begin if pc>=pd then trap(EBADPC); nextpc:=code[pc]; newpc(pc+1) end;

procedure lino(w:word);
```

```
begin store(lineadr,w) end;

procedure filna(a:adr);
begin storea(fileadr,a) end;


{---------------------------------------------------------------------------}
{                    Stack Manipulation Routines                            }
{---------------------------------------------------------------------------}

{ push puts a word on the stack
  pushsw takes a signed one word integer and pushes it on the stack
  pop removes a machine word from the stack and delivers it as a word
  popsw removes a machine word from the stack and delivers a signed integer
  pusha pushes an address on the stack
  popa removes a machine word from the stack and delivers it as an address
  pushd pushes a double precision number on the stack
  popd removes two machine words and returns a double precision integer
  pushr pushes a float (floating point) number on the stack
  popr removes several machine words and returns a float number
  pushx puts an object of arbitrary size on the stack
  popx removes an object of arbitrary size
                                                                          }

procedure push(x:word);
begin newsp(sp-wsize); store(sp,x) end;

procedure pushsw(x:sword);
begin newsp(sp-wsize); store(sp,unsign(x)) end;

function pop:word;
begin pop:=memw(sp); newsp(sp+wsize) end;

function popsw:sword;
begin popsw:=signwd(pop) end;

procedure pusha(x:adr);
begin newsp(sp-asize); storea(sp,x) end;

function popa:adr;
begin popa:=mema(sp); newsp(sp+asize) end;

procedure pushd(y:double);
begin { push double integer onto the stack } newsp(sp-2*wsize) end;

function popd:double;
begin { pop double integer from the stack } newsp(sp+2*wsize); popd:=0 end;

procedure pushr(z:real);
begin { Push a float onto the stack } newsp(sp-fsize) end;

function popr:real;
begin { pop float from the stack } newsp(sp+fsize); popr:=0.0 end;

procedure pushx(objsize:size; a:adr);
var i:integer;
begin
  if objsize<wsize
     then push(mems(a,objsize))
```

```
     else for i:=1 to objsize div wsize do push(memw(a+objsize-wsize*i))
end;

procedure popx(objsize:size; a:adr);
var i:integer;
begin
  if objsize<wsize
     then stores(a,objsize,pop)
     else for i:=1 to objsize div wsize do store(a-wsize+wsize*i,pop)
end;


{---------------------------------------------------------------------------}
{              Bit manipulation routines (extract, shift, rotate)           }
{---------------------------------------------------------------------------}

procedure sleft(var w:sword);  { 1 bit left shift   }
begin w:= dosign(fitsw(2*w,EIOVFL)) end;

procedure suleft(var w:word);  { 1 bit left shift   }
begin w := chopw(2*w) end;

procedure sdleft(var d:double);  { 1 bit left shift   }
begin { shift two word signed integer } end;

procedure sright(var w:sword);  { 1 bit right shift with sign extension }
begin if w >= 0 then w := w div 2 else w := (w-1) div 2 end;

procedure suright(var w:word);    { 1 bit right shift without sign extension }
begin w := w div 2 end;

procedure sdright(var d:double);  { 1 bit right shift   }
begin { shift two word signed integer } end;

procedure rleft(var w:word);  { 1 bit left rotate }
begin if w >= t15
        then w:=(w-t15)*2 + 1
        else w:=w*2
end;

procedure rright(var w:word);  { 1 bit right rotate }
begin if w mod 2 = 1
        then w:=w div 2 + t15
        else w:=w div 2
end;

function sextend(w:word;s:size):word;
var i:size;
begin
  for i:=1 to (wsize-s)*8 do rleft(w);
  for i:=1 to (wsize-s)*8 do sright(w);
  sextend:=w;
end;

function bit(b:bitnr; w:word):bitval; { return bit b of the word w }
var i:bitnr;
begin for i:= 1 to b do rright(w); bit:= w mod 2 end;

function bf(ty:bftype; w1,w2:word):word;  { return boolean fcn of 2 words }
```

```
var i:bitnr; j:word;
begin j:=0;
      for i:= maxbitnr downto 0 do
        begin j := 2*j;
              case ty of
                andf: if bit(i,w1)+bit(i,w2) = 2 then j:=j+1;
                iorf: if bit(i,w1)+bit(i,w2) > 0 then j:=j+1;
                xorf: if bit(i,w1)+bit(i,w2) = 1 then j:=j+1
              end
        end;
      bf:=j
end;

{---------------------------------------------------------------------------}
{                            Array indexing                                  }
{---------------------------------------------------------------------------}

function arraycalc(c:adr):adr; { subscript calculation }
var j:full; objsize:size; a:adr;
begin j:= popsw - signwd(memw(c));
  if (j<0) or (j>memw(c+wsize)) then trap(EARRAY);
  objsize := argo(memw(c+wsize+wsize));
  a := j*objsize+popa; chkadr(a,objsize);
  arraycalc:=a
end;


{---------------------------------------------------------------------------}
{                        Double and Real Arithmetic                          }
{---------------------------------------------------------------------------}

{ All routines for doubles and floats are dummy routines, since the format of
  doubles and floats is not defined in EM.
}

function doadi(ds,dt:double):double;
begin { add two doubles } doadi:=0 end;

function dosbi(ds,dt:double):double;
begin { subtract two doubles } dosbi:=0 end;

function domli(ds,dt:double):double;
begin { multiply two doubles } domli:=0 end;

function dodvi(ds,dt:double):double;
begin { divide two doubles } dodvi:=0 end;

function dormi(ds,dt:double):double;
begin { modulo of two doubles } dormi:=0 end;

function dongi(ds:double):double;
begin { negative of a double } dongi:=0 end;

function doadf(x,y:real):real;
begin { add two floats } doadf:=0.0 end;

function dosbf(x,y:real):real;
begin { subtract two floats } dosbf:=0.0 end;
```

```
function domlf(x,y:real):real;
begin { multiply two floats } domlf:=0.0 end;

function dodvf(x,y:real):real;
begin { divide two floats } dodvf:=0.0 end;

function dongf(x:real):real;
begin { negate a float } dongf:=0.0 end;

procedure dofif(x,y:real;var intpart,fraction:real);
begin { dismember x*y into integer and fractional parts }
  intpart:=0.0;  { integer part of x*y, same sign as x*y }
  fraction:=0.0;
        { fractional part of x*y, 0<=abs(fraction)<1 and same sign as x*y }
end;

procedure dofef(x:real;var mantissa:real;var exponent:sword);
begin { dismember x into mantissa and exponent parts }
  mantissa:=0.0;  { mantissa of x , >= 1/2 and <1 }
  exponent:=0;    { base 2 exponent of x }
end;
```

```
{---------------------------------------------------------------------------}
{                            Trap and Call                                  }
{---------------------------------------------------------------------------}

procedure call(p:adr); { Perform the call }
begin
  pusha(lb);pusha(pc);
  newlb(sp);newsp(sp - memi(pd + pdsize*p + pdlocs));
  newpc(memi(pd + pdsize*p+ pdbase))
end;

procedure dotrap(n:byte);
var i:size;
begin
  if (uerrorproc=0) or intrap then
    begin
      if intrap then
        writeln('Recursive trap, first trap number was ', trapval:1);
      writeln('Error ', n:1);
      writeln('With',ord(insr):4,' arg ',k:1);
      goto 9999
    end;
  { Deposit all interpreter variables that need to be saved on
    the stack. This includes all scratch variables that can
    be in use at the moment and ( not possible in this interpreter )
    the internal address of the interpreter where the error occurred.
    This would make it possible to execute an RTT instruction totally
    transparent to the user program.
    It can, for example, occur within an ADD instruction that both
    operands are undefined and that the result overflows.
    Although this will generate 3 error traps it must be possible
    to ignore them all.
}
  intrap:=true; trapval:=n;
  for i:=retsize div wsize downto 1 do push(retarea[i]);
  push(retsize);              { saved return area }
  pusha(mema(fileadr));       { saved current file name pointer }
  push(memw(lineadr));        { saved line number }
  push(n);                    { push error number }
  a:=argp(uerrorproc);
  uerrorproc:=0;              { reset signal }
  call(a);                    { call the routine }
  intrap:=false;              { Don't catch recursive traps anymore }
  goto 8888;                  { reenter main loop }
end;

procedure trap;
{ This routine is invoked for overflow, and other run time errors.
  For non-fatal errors, trap returns to the calling routine
}
begin
  if n>=16 then dotrap(n) else if bit(n,ignmask)=0 then dotrap(n);
end;

procedure dortt;
{ The restoration of file address and line number is not essential.
  The restoration of the return save area is.
}
var i:size;
```

```
    n:word;
begin
  newsp(lb); lb:=maxdata+1 ; { to circumvent ESTACK for the popa + pop }
  newpc(popa); newlb(popa); { So far a plain RET 0 }
  n:=pop; if (n>=16) and (n<64) then goto 9999 ;
  lino(pop); filna(popa); retsize:=pop;
  for i:=1 to retsize div wsize do retarea[i]:=pop ;
end;


{---------------------------------------------------------------------------}
{                              monitor calls                                }
{---------------------------------------------------------------------------}


procedure domon(entry:word);
var     index:  1..63;
        dummy:  double;
        count,rwptr:    adr;
        token:  byte;
        i:      integer;
begin
  if (entry<=0) or (entry>63) then entry:=63 ;
  index:=entry;
  case index of
   1: begin { exit } exitstatus:=pop; halted:=true end;
   3: begin { read }  dummy:=pop; { All input is from stdin }
        rwptr:=popa; count:=popa;
        i:=0 ;
        while (not eof(input)) and (i<count) do
        begin
          if eoln(input) then begin storeb(rwptr,10) ; count:=i end
                         else storeb(rwptr,ord(input^)) ;
          get(input); rwptr:=rwptr+1 ; i:=i+1 ;
        end;
        pusha(i); push(0)
      end;
   4: begin { write } dummy:=pop; { All output is to stdout }
        rwptr:=popa; count:=popa;
        for i:=1 to count do
          begin token:=memb(rwptr); rwptr:=rwptr+1 ;
            if token=10 then writeln else write(chr(token))
          end ;
        pusha(count);
        push(0)
      end;
  54: begin { ioctl, faked } dummy:=popa;dummy:=popa;dummy:=pop;push(0) end ;
       2,          5,  6,  7,  8,  9, 10,
  11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
  21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
  31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
  41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
  51, 52, 53,     55, 56, 57, 58, 59, 60,
  61, 62:
      begin push(22); push(22) end;
  63: { exists only for the trap }
      trap(EBADMON)
  end
end;
```

```
{---------------------------------------------------------------------------}
{                        Initialization and debugging                       }
{---------------------------------------------------------------------------}

procedure doident; { print line number and file name }
var a:adr; i,c:integer; found:boolean;
begin
  write('at line ',memw(lineadr):1,' ');
  a:=mema(fileadr); if a<>0 then
  begin i:=20; found:=false;
    while (i<>0) and not found do
    begin c:=memb(a); a:=a+1; found:=true; i:=i-1;
      if (c>=48) and (c<=57) then
        begin found:=false; write(chr(ord('0')+c-48)) end;
      if (c>=65) and (c<=90) then
        begin found:=false; write(chr(ord('A')+c-65)) end;
      if (c>=97) and (c<=122) then
        begin found:=false; write(chr(ord('a')+c-97)) end;
    end;
  end;
  writeln;
end;

procedure initialize;  { start the ball rolling }
{ This is not part of the machine definition }
var cset:set of char;
    f:ifset;
    iclass:insclass;
    insno:byte;
    nops:integer;
    opcode:byte;
    i,j,n:integer;
    wtemp:sword;
    count:integer;
    repc:adr;
    nexta,firsta:adr;
    elem:byte;
    amount,ofst:size;
    c:char;

    function readb(n:integer):double;
    var b:byte;
    begin read(prog,b); if n>1 then readb:=readb(n-1)*256+b else readb:=b end;

    function readbyte:byte;
    begin readbyte:=readb(1) end;

    function readword:word;
    begin readword:=readb(wsize) end;

    function readadr:adr;
    begin readadr:=readb(asize) end;

    function ifind(ordinal:byte):mnem;
    var loopvar:mnem;
        found:boolean;
    begin ifind:=NON;
      loopvar:=insr; found:=false;
      repeat
```

```
        if ordinal=ord(loopvar) then
          begin found:=true; ifind:=loopvar end;
        if loopvar<>ZRL then loopvar:=succ(loopvar) else loopvar:=NON;
      until found or (loopvar=insr) ;
   end;

    procedure readhdr;
    type hdrw=0..32767 ; { 16 bit header words }
    var  hdr: hdrw;
         i: integer;
    begin
      for i:=0 to 7 do
      begin hdr:=readb(2);
        case i of
        0: if hdr<>3757 then { 07255 }
           begin writeln('Not an em load file'); halt end;
        2: if hdr<>0 then
           begin writeln('Unsolved references'); halt end;
        3: if hdr<>3 then
           begin writeln('Incorrect load file version'); halt end;
        4: if hdr<>wsize then
           begin writeln('Incorrect word size'); halt end;
        5: if hdr<>asize then
           begin writeln('Incorrect pointer size'); halt end;
        1,6,7:;
        end
      end
    end;

    procedure noinit;
    begin writeln('Illegal initialization'); halt end;

    procedure readint(a:adr;s:size);
    var i:size;
    begin { construct integer out of byte sequence }
      for i:=1 to s do { construct the value and initialize at a }
        begin storeb(a,readbyte); a:=a+1 end
    end;

    procedure readuns(a:adr;s:size);
    begin { construct unsigned out of byte sequence }
      readint(a,s) { identical to readint }
    end;

    procedure readfloat(a:adr;s:size);
    var i:size; b:byte;
    begin { construct float out of string}
      if (s<>4) and (s<>8) then noinit; i:=0;
      repeat { eat the bytes, construct the value and intialize at a }
        b:=readbyte; i:=i+1;
      until b=0 ;
    end;

begin
  halted:=false;
  exitstatus:=undef;
  uerrorproc:=0; intrap:=false;

  { initialize tables }
```

```
for i:=0 to maxcode do code[i]:=0;
for i:=0 to maxdata do data[i]:=0;
for iclass:=prim to tert do
  for i:=0 to 255 do
    with dispat[iclass][i] do
      begin instr:=NON; iflag:=[zbit] end;

{ read instruction table file. see appendix B }
{ The table read here is a simple transformation of the table on page xx }
{ - instruction names were transformed to numbers }
{ - the '-' flag was transformed to an 'i' flag for 'w' type instructions }
{ - the 'S' flag was added for instructions having signed operands }
reset(tables);
insr:=NON;
repeat
  read(tables,insno) ; cset:=[]; f:=[];
  insr:=ifind(insno);
  if insr=NON then begin writeln('Incorrect table'); halt end;
  repeat read(tables,c) until c<>' ' ;
  repeat
    cset:=cset+[c];
    read(tables,c)
  until c=' ' ;
  if 'm' in cset then f:=f+[mini];
  if 's' in cset then f:=f+[short];
  if '-' in cset then f:=f+[zbit];
  if 'i' in cset then f:=f+[ibit];
  if 'S' in cset then f:=f+[sbit];
  if 'w' in cset then f:=f+[wbit];
  if (mini in f) or (short in f) then read(tables,nops) else nops:=1 ;
  readln(tables,opcode);
  if ('4' in cset) or ('8' in cset) then
    begin iclass:=tert end
  else if 'e' in cset then
    begin iclass:=second end
  else iclass:=prim;
  for i:=0 to nops-1 do
  begin
    with dispat[iclass,opcode+i] do
    begin
      iflag:=f; instr:=insr;
      if '2' in cset      then ilength:=2
      else if 'u' in cset then ilength:=2
      else if '4' in cset then ilength:=4
      else if '8' in cset then ilength:=8
      else if (mini in f) or (short in f) then
        begin
          if 'N' in cset then wtemp:=-1-i else wtemp:=i ;
          if 'o' in cset then wtemp:=wtemp+1 ;
          if short in f then wtemp:=wtemp*256 ;
          implicit:=wtemp
        end
    end
  end
until eof(tables);

{ read in program text, data and procedure descriptors }
reset(prog);
readhdr;                               { verify first header }
```

```
for i:=1 to 8 do header[i]:=readadr;  { read second header }
hp:=maxdata+1; sp:=maxdata+1; lino(0);
{ read program text }
if header[NTEXT]+header[NPROC]*pdsize>maxcode then
  begin writeln('Text size too large'); halt end;
if header[SZDATA]>maxdata then
  begin writeln('Data size too large'); halt end;
for i:=0 to header[NTEXT]-1 do code[i]:=readbyte;
{ read data blocks }
nexta:=0;
for i:=1 to header[NDATA] do
  begin
    n:=readbyte;
    if n<>0 then
      begin
        elem:=readbyte; firsta:=nexta;
        case n of
        1: { uninitialized words }
           for j:=1 to elem do
           begin store(nexta,undef); nexta:=nexta+wsize end;
        2: { initialized bytes }
           for j:=1 to elem do
           begin storeb(nexta,readbyte); nexta:=nexta+1 end;
        3: { initialized words }
           for j:=1 to elem do
           begin store(nexta,readword); nexta:=nexta+wsize end;
        4,5: { instruction and data pointers }
           for j:=1 to elem do
           begin storea(nexta,readadr); nexta:=nexta+asize end;
        6: { signed integers }
           begin readint(nexta,elem); nexta:=nexta+elem end;
        7: { unsigned integers }
           begin readuns(nexta,elem); nexta:=nexta+elem end;
        8: { floating point numbers }
           begin readfloat(nexta,elem); nexta:=nexta+elem end;
        end
      end
    else
      begin
        repc:=readadr; amount:=nexta-firsta;
        for count:=1 to repc do
        begin
          for ofst:=0 to amount-1 do data[nexta+ofst]:=data[firsta+ofst];
          nexta:=nexta+amount;
        end
      end
  end;
if header[SZDATA]<>nexta then writeln('Data initialization error');
hp:=nexta;
{ read descriptor table }
pd:=header[NTEXT];
for i:=1 to header[NPROC]*pdsize do code[pd+i-1]:=readbyte;
{ call the entry point routine }
ignmask:=0;      { catch all traps, higher numbered traps cannot be ignored}
retsize:=0;
lb:=maxdata;     { illegal dynamic link }
pc:=maxcode;     { illegal return address }
push(0); a:=sp;  { No environment }
push(0); b:=sp;  { No args }
```

```
  pusha(a);        { envp }
  pusha(b);        { argv }
  push(0);         { argc }
  call(argp(header[ENTRY]));
end;
```

```
{---------------------------------------------------------------------------}
{                       MAIN LOOP OF THE INTERPRETER                        }
{---------------------------------------------------------------------------}
{ It should be noted that the interpreter (microprogram) for  an  EM
  machine  can be written in two fundamentally different ways: (1) the
  instruction operands are fetched in the main loop, or  (2)  the  in-
  struction operands are fetched after the 256 way branch, by the exe-
  cution routines themselves.  In this interpreter, method (1) is used
  to  simplify  the  description  of execution  routines. The dispatch
  table dispat is used to determine how the operand is  encoded. There
  are 4 possibilities:

     0. There is no operand
     1. The operand and  instruction are  together in 1  byte (mini)
     2. The operand is  one byte long and follows the opcode byte(s)
     3. The operand is two bytes long and follows the opcode byte(s)
     4. The operand is four bytes long and follows the opcode byte(s)

  In  this  interpreter,  the  main  loop determines the operand type,
  fetches it, and leaves it in the global variable k for the execution
  routines  to use.  Consequently, instructions such as LOL, which use
  three different formats, need only be described once in the body  of
  the interpreter.
      However, for  a  production  interpreter,  or  a  hardware  EM
  machine,  it  is  probably better to use method (2), i.e. to let the
  execution routines themselves fetch their own operands.  The  reason
  for this is that each opcode uniquely determines the operand format,
  so no table lookup in the dispatch table is needed.  The whole table
  is not needed. Method (2) therefore executes much faster.
      However, separate execution routines will be needed for LOL with
  a one byte offset,  and  LOL with a two byte offset.  It is to avoid
  this additional clutter that method (1) is used here.  In a  produc-
  tion interpreter, it is envisioned that the main loop will fetch the
  next instruction byte, and use it as an index into a 256 word  table
  to  find  the  address  of  the interpreter routine to jump to.  The
  routine jumped to will  begin  by  fetching  its  operand,  if  any,
  without  any  table  lookup,  since it knows which format to expect.
  After doing the work, it returns to the main  loop  by  jumping  in-
  directly  to  a register that contains the address of the main loop.
      A slight variation on this idea is to have the  register contain
  the address of the branch table, rather than the address of the main
  loop.
      Another  issue  is whether the execution routines for LOL 0, LOL
  2, LOL 4, etc. should all be have distinct execution routines. Doing
  so  provides for the maximum speed, since the operand is implicit in
  the routine itself.  The disadvantage is that many nearly  identical
  execution  routines will then be needed.  Another way of doing it is
  to keep the instruction byte fetched from memory (LOL 0, LOL 2,  LOL
  4, etc.) in some register, and have all the LOL mini format instruc-
  tions branch to a common routine.  This routine can  then  determine
  the  operand  by  subtracting  the code for LOL 0 from the register,
  leaving the true operand in the register  (as  a  word  quantity  of
  course).   This  method  makes the interpreter smaller, but is a bit
  slower.
```

To make this important point a little clearer, consider how a production interpreter for the PDP-11 might appear. Let us assume the following opcodes have been assigned:

```
31: LOL -2     (2 bytes, i.e. next word)
32: LOL -4
33: LOL -6
34: LOL b      (format with a one byte offset)
35: LOL w      (format with a one word, i.e. two byte offset)
```

Further assume that each of the 5 opcodes will have its own execution routine, i.e. we are making a tradeoff in favor of fast execution and a slightly larger interpreter.

Register r5 is the em program counter.
Register r4 is the em LB register
Register r3 is the em SP register (the stack grows toward low core)
Register r2 contains the interpreter address of the main loop

The main loop looks like this:

```
movb (r5)+,r0          /fetch the opcode into r0 and increment r5
asl r0                 /shift r0 left 1 bit. Now: -256<=r0<=+254
jmp *table(r0)         /jump to execution routine
```

Notice that no operand fetching has been done. The execution routines for the 5 sample instructions given above might be as follows:

```
lol2: mov -2(r4),-(sp)      /push local -2 onto stack
      jmp (r2)              /go back to main loop
lol4: mov -4(r4),-(sp)      /push local -4 onto stack
      jmp (r2)              /go back to main loop
lol6: mov -6(r4),-(sp)      /push local -6 onto stack
      jmp (r2)              /go back to main loop
lolb: mov $177400,r0        /prepare to fetch the 1 byte operand
      bisb (r5)+,r0         /operand is now in r0
      asl r0                /r0 is now offset from LB in bytes, not words
      add r4,r0             /r0 is now address of the needed local
      mov (r0),-(sp)        /push the local onto the stack
      jmp (r2)
lolw: clr r0                /prepare to fetch the 2 byte operand
      bisb (r5)+,r0         /fetch high order byte first !!!
      swab r0               /insert high order byte in place
      bisb (r5)+,r0         /insert low order byte in place
      asl r0                /convert offset to bytes, from words
      add r4,r0             /r0 is now address of needed local
      mov (r0),-(sp)        /stack the local
      jmp (r2)              /done
```

The important thing to notice is where and how the operand fetch occurred:

lol2, lol4, and lol6, (the mini’s) have implicit operands
lolb knew it had to fetch one byte, and did so without any table lookup
lolw knew it had to fetch a word, and did so, high order byte first }

```
{---------------------------------------------------------------------------}
{           Routines for the individual instructions                        }
{---------------------------------------------------------------------------}
procedure loadops;
var j:integer;
begin
  case insr of
    { LOAD GROUP }
    LDC: pushd(argd(k));
    LOC: pushsw(argc(k));
    LOL: push(memw(locadr(k)));
    LOE: push(memw(argg(k)));
    LIL: push(memw(mema(locadr(k))));
    LOF: push(memw(popa+argf(k)));
    LAL: pusha(locadr(k));
    LAE: pusha(argg(k));
    LXL: begin a:=lb; for j:=1 to argn(k) do a:=mema(a+savsize); pusha(a) end;
    LXA: begin a:=lb;
           for j:=1 to argn(k) do a:= mema(a+savsize);
           pusha(a+savsize)
         end;
    LOI: pushx(argo(k),popa);
    LOS: begin k:=argw(k); if k<>wsize then trap(EILLINS);
           k:=pop; pushx(argo(k),popa)
         end;
    LDL: begin a:=locadr(k); push(memw(a+wsize)); push(memw(a)) end;
    LDE: begin k:=argg(k); push(memw(k+wsize)); push(memw(k)) end;
    LDF: begin k:=argf(k);
           a:=popa; push(memw(a+k+wsize)); push(memw(a+k))
         end;
    LPI: push(argp(k))
  end
end;

procedure storeops;
begin
  case insr of
    { STORE GROUP }
    STL: store(locadr(k),pop);
    STE: store(argg(k),pop);
    SIL: store(mema(locadr(k)),pop);
    STF: begin a:=popa; store(a+argf(k),pop) end;
    STI: popx(argo(k),popa);
    STS: begin k:=argw(k); if k<>wsize then trap(EILLINS);
           k:=popa; popx(argo(k),popa)
         end;
    SDL: begin a:=locadr(k); store(a,pop); store(a+wsize,pop) end;
    SDE: begin k:=argg(k); store(k,pop); store(k+wsize,pop) end;
    SDF: begin k:=argf(k); a:=popa; store(a+k,pop); store(a+k+wsize,pop) end
  end
end;

procedure intarith;
var i:integer;
begin
  case insr of
    { SIGNED INTEGER ARITHMETIC }
    ADI: case szindex(argw(k)) of
         1: begin st:=popsw; ss:=popsw; push(fitsw(ss+st,EIOVFL)) end;
```

```
          2: begin dt:=popd; ds:=popd; pushd(doadi(ds,dt)) end;
          end ;
     SBI: case szindex(argw(k)) of
          1: begin st:=popsw; ss:= popsw; push(fitsw(ss-st,EIOVFL)) end;
          2: begin dt:=popd; ds:=popd; pushd(dosbi(ds,dt)) end;
          end ;
     MLI: case szindex(argw(k)) of
          1: begin st:=popsw; ss:= popsw; push(fitsw(ss*st,EIOVFL)) end;
          2: begin dt:=popd; ds:=popd; pushd(domli(ds,dt)) end;
          end ;
     DVI: case szindex(argw(k)) of
          1: begin st:= popsw; ss:= popsw;
                if st=0 then trap(EIDIVZ) else pushsw(ss div st)
             end;
          2: begin dt:=popd; ds:=popd; pushd(dodvi(ds,dt)) end;
          end;
     RMI: case szindex(argw(k)) of
          1: begin st:= popsw; ss:=popsw;
                if st=0 then trap(EIDIVZ) else pushsw(ss - (ss div st)*st)
             end;
          2: begin dt:=popd; ds:=popd; pushd(dormi(ds,dt)) end
          end;
     NGI: case szindex(argw(k)) of
          1: begin st:=popsw; pushsw(-st) end;
          2: begin ds:=popd; pushd(dongi(ds)) end
          end;
     SLI: begin t:=pop;
            case szindex(argw(k)) of
            1: begin ss:=popsw;
                  for i:= 1 to t do sleft(ss); pushsw(ss)
               end
            end
          end;
     SRI: begin t:=pop;
            case szindex(argw(k)) of
            1: begin ss:=popsw;
                  for i:= 1 to t do sright(ss); pushsw(ss)
               end;
            2: begin ds:=popd;
                  for i:= 1 to t do sdright(ss); pushd(ss)
               end
            end
          end
  end
end;

procedure unsarith;
var i:integer;
begin
  case insr of
    { UNSIGNED INTEGER ARITHMETIC }
    ADU: case szindex(argw(k)) of
          1: begin t:=pop; s:= pop; push(chopw(s+t)) end;
          2: trap(EILLINS);
          end ;
    SBU: case szindex(argw(k)) of
          1: begin t:=pop; s:= pop; push(chopw(s-t)) end;
          2: trap(EILLINS);
          end ;
```

```
    MLU: case szindex(argw(k)) of
         1: begin t:=pop; s:= pop; push(chopw(s*t)) end;
         2: trap(EILLINS);
         end ;
    DVU: case szindex(argw(k)) of
         1: begin t:= pop; s:= pop;
                if t=0 then trap(EIDIVZ) else push(s div t)
            end;
         2: trap(EILLINS);
         end;
    RMU: case szindex(argw(k)) of
         1: begin t:= pop; s:=pop;
                if t=0 then trap(EIDIVZ) else push(s - (s div t)*t)
            end;
         2: trap(EILLINS);
         end;
    SLU: case szindex(argw(k)) of
         1: begin t:=pop; s:=pop;
                for i:= 1 to t do suleft(s); push(s)
            end;
         2: trap(EILLINS);
         end;
    SRU: case szindex(argw(k)) of
         1: begin t:=pop; s:=pop;
                for i:= 1 to t do suright(s); push(s)
            end;
         2: trap(EILLINS);
         end
  end
end;

procedure fltarith;
begin
  case insr of
    { FLOATING POINT ARITHMETIC }
    ADF: begin argwf(k); rt:=popr; rs:=popr; pushr(doadf(rs,rt)) end;
    SBF: begin argwf(k); rt:=popr; rs:=popr; pushr(dosbf(rs,rt)) end;
    MLF: begin argwf(k); rt:=popr; rs:=popr; pushr(domlf(rs,rt)) end;
    DVF: begin argwf(k); rt:=popr; rs:=popr; pushr(dodvf(rs,rt)) end;
    NGF: begin argwf(k); rt:=popr; pushr(dongf(rt)) end;
    FIF: begin argwf(k); rt:=popr; rs:=popr;
                dofif(rt,rs,x,y); pushr(y); pushr(x)
         end;
    FEF: begin argwf(k); rt:=popr; dofef(rt,x,ss); pushr(x); pushsw(ss) end
  end
end;

procedure ptrarith;
begin
  case insr of
    { POINTER ARITHMETIC }
    ADP: pusha(popa+argf(k));
    ADS: case szindex(argw(k)) of
         1: begin st:=popsw; pusha(popa+st) end;
         2: begin dt:=popd; pusha(popa+dt) end;
         end;
    SBS: begin
           a:=popa; b:=popa;
           case szindex(argw(k)) of
```

```
            1: push(fitsw(b-a,EIOVFL));
            2: pushd(b-a)
          end
        end
  end
end;

procedure incops;
var j:integer;
begin
  case insr of
    { INCREMENT/DECREMENT/ZERO }
    INC: push(fitsw(popsw+1,EIOVFL));
    INL: begin a:=locadr(k); store(a,fitsw(signwd(memw(a))+1,EIOVFL)) end;
    INE: begin a:=argg(k); store(a,fitsw(signwd(memw(a))+1,EIOVFL)) end;
    DEC: push(fitsw(popsw-1,EIOVFL));
    DEL: begin a:=locadr(k); store(a,fitsw(signwd(memw(a))-1,EIOVFL)) end;
    DEE: begin a:=argg(k); store(a,fitsw(signwd(memw(a))-1,EIOVFL)) end;
    ZRL: store(locadr(k),0);
    ZRE: store(argg(k),0);
    ZER: for j:=1 to argw(k) div wsize do push(0);
    ZRF: pushr(0);
  end
end;

procedure convops;
begin
  case insr of
    { CONVERT GROUP }
    CII: begin s:=pop; t:=pop;
           if t<wsize then begin push(sextend(pop,t)); t:=wsize end;
           case szindex(argw(t)) of
           1: if szindex(argw(s))=2 then pushd(popsw);
           2: if szindex(argw(s))=1 then push(fitsw(popd,ECONV))
           end
         end;
    CIU: case szindex(argw(pop)) of
         1: if szindex(argw(pop))=2 then push(unsign(popd mod negoff));
         2: trap(EILLINS);
         end;
    CIF: begin argwf(pop);
           case szindex(argw(pop)) of 1:pushr(popsw); 2:pushr(popd) end
         end;
    CUI: case szindex(argw(pop)) of
         1: case szindex(argw(pop)) of
              1: begin s:=pop; if s>maxsint then trap(ECONV); push(s) end;
              2: trap(EILLINS);
              end;
         2: case szindex(argw(pop)) of
              1: pushd(pop);
              2: trap(EILLINS);
              end;
         end;
    CUU: case szindex(argw(pop)) of
         1: if szindex(argw(pop))=2 then trap(EILLINS);
         2: trap(EILLINS);
         end;
    CUF: begin argwf(pop);
           if szindex(argw(pop))=1 then pushr(pop) else trap(EILLINS)
```

```
        end;
    CFI: begin sz:=argw(pop); argwf(pop); rt:=popr;
          case szindex(sz) of
          1: push(fitsw(trunc(rt),ECONV));
          2: pushd(fitd(trunc(rt)));
          end
        end;
    CFU: begin sz:=argw(pop); argwf(pop); rt:=popr;
          case szindex(sz) of
          1: push( chopw(trunc(abs(rt)-0.5)) );
          2: trap(EILLINS);
          end
        end;
    CFF: begin argwf(pop); argwf(pop) end
  end
end;

procedure logops;
var i,j:integer;
begin
  case insr of
    { LOGICAL GROUP }
    XAND:
        begin k:=argw(k);
          for j:= 1 to k div wsize do
            begin a:=sp+k; t:=pop; store(a,bf(andf,memw(a),t)) end;
        end;
    IOR:
        begin k:=argw(k);
          for j:= 1 to k div wsize do
            begin a:=sp+k; t:=pop; store(a,bf(iorf,memw(a),t)) end;
        end;
    XOR:
        begin k:=argw(k);
          for j:= 1 to k div wsize do
            begin a:=sp+k; t:=pop; store(a,bf(xorf,memw(a),t)) end;
        end;
    COM:
        begin k:=argw(k);
          for j:= 1 to k div wsize do
            begin
              store(sp+k-wsize*j, bf(xorf,memw(sp+k-wsize*j), negoff-1))
            end
        end;
    ROL: begin k:=argw(k); if k<>wsize then trap(EILLINS);
          t:=pop; s:=pop; for i:= 1 to t do rleft(s); push(s)
        end;
    ROR: begin k:=argw(k); if k<>wsize then trap(EILLINS);
          t:=pop; s:=pop; for i:= 1 to t do rright(s); push(s)
        end
  end
end;

procedure setops;
var i,j:integer;
begin
  case insr of
    { SET GROUP }
    INN:
```

```
         begin k:=argw(k);
           t:=pop;
           i:= t mod 8; t:= t div 8;
           if t>=k then
             begin trap(ESET); s:=0 end
           else
             begin s:=memb(sp+t) end;
           newsp(sp+k); push(bit(i,s));
         end;
    XSET:
         begin k:=argw(k);
           t:=pop;
           i:= t mod 8; t:= t div 8;
           for j:= 1 to k div wsize do push(0);
           if t>=k then
             trap(ESET)
           else
             begin s:=1; for j:= 1 to i do rleft(s); storeb(sp+t,s) end
         end
  end
end;

procedure arrops;
begin
  case insr of
    { ARRAY GROUP }
    LAR:
         begin k:=argw(k); if k<>wsize then trap(EILLINS); a:=popa;
           pushx(argo(memw(a+2*k)),arraycalc(a))
         end;
    SAR:
         begin k:=argw(k); if k<>wsize then trap(EILLINS); a:=popa;
           popx(argo(memw(a+2*k)),arraycalc(a))
         end;
    AAR:
         begin k:=argw(k); if k<>wsize then trap(EILLINS); a:=popa;
           push(arraycalc(a))
         end
  end
end;

procedure cmpops;
begin
  case insr of
    { COMPARE GROUP }
    CMI: case szindex(argw(k)) of
         1: begin st:=popsw; ss:=popsw;
              if ss<st then pushsw(-1) else if ss=st then push(0) else push(1)
            end;
         2: begin dt:=popd; ds:=popd;
              if ds<dt then pushsw(-1) else if ds=dt then push(0) else push(1)
            end;
         end;
    CMU: case szindex(argw(k)) of
         1: begin t:=pop; s:=pop;
              if s<t then pushsw(-1) else if s=t then push(0) else push(1)
            end;
         2: trap(EILLINS);
         end;
```

```
    CMP: begin a:=popa; b:=popa;
          if b<a then pushsw(-1) else if b=a then push(0) else push(1)
         end;
    CMF: begin argwf(k); rt:=popr; rs:=popr;
           if rs<rt then pushsw(-1) else if rs=rt then push(0) else push(1)
         end;
    CMS: begin k:=argw(k);
           t:= 0; j:= 0;
           while (j < k) and (t=0) do
             begin if memw(sp+j) <> memw(sp+k+j) then t:=1;
               j:=j+wsize
             end;
           newsp(sp+wsize*k); push(t);
         end;

    TLT: if popsw <  0 then push(1) else push(0);
    TLE: if popsw <= 0 then push(1) else push(0);
    TEQ: if pop   =  0 then push(1) else push(0);
    TNE: if pop   <> 0 then push(1) else push(0);
    TGE: if popsw >= 0 then push(1) else push(0);
    TGT: if popsw >  0 then push(1) else push(0);
  end
end;

procedure branchops;
begin
  case insr of
    { BRANCH GROUP }
    BRA: newpc(pc+k);

    BLT: begin st:=popsw; if popsw <  st then newpc(pc+k) end;
    BLE: begin st:=popsw; if popsw <= st then newpc(pc+k) end;
    BEQ: begin t :=pop  ; if pop   =   t then newpc(pc+k) end;
    BNE: begin t :=pop  ; if pop   <>  t then newpc(pc+k) end;
    BGE: begin st:=popsw; if popsw >= st then newpc(pc+k) end;
    BGT: begin st:=popsw; if popsw >  st then newpc(pc+k) end;

    ZLT: if popsw <  0 then newpc(pc+k);
    ZLE: if popsw <= 0 then newpc(pc+k);
    ZEQ: if pop   =  0 then newpc(pc+k);
    ZNE: if pop   <> 0 then newpc(pc+k);
    ZGE: if popsw >= 0 then newpc(pc+k);
    ZGT: if popsw >  0 then newpc(pc+k)
  end
end;

procedure callops;
var j:integer;
begin
  case insr of
    { PROCEDURE CALL GROUP }
    CAL: call(argp(k));
    CAI: begin call(argp(popa)) end;
    RET: begin k:=argz(k); if k div wsize>maxret  then trap(EILLINS);
           for j:= 1 to k div wsize do retarea[j]:=pop; retsize:=k;
           newsp(lb); lb:=maxdata+1; { To circumvent stack overflow error }
           newpc(popa);
           if pc=maxcode then
           begin
```

```
            halted:=true;
            if retsize=wsize then exitstatus:=retarea[1]
              else exitstatus:=undef
         end
         else
            newlb(popa);
        end;
    LFR: begin k:=args(k); if k<>retsize then trap(EILLINS);
          for j:=k div wsize downto 1 do push(retarea[j]);
        end
  end
end;

procedure miscops;
var i,j:integer;
begin
  case insr of
    { MISCELLANEOUS GROUP }
    ASP,ASS:
        begin if insr=ASS then
          begin k:=argw(k); if k<>wsize then trap(EILLINS); k:=popsw end;
          k:=argf(k);
          if k<0
            then for j:= 1 to -k div wsize do push(undef)
            else newsp(sp+k);
        end;
    BLM,BLS:
        begin if insr=BLS then
          begin k:=argw(k); if k<>wsize then trap(EILLINS); k:=pop end;
          k:=argz(k);
          b:=popa; a:=popa;
          for j := 1 to k div wsize do
            store(b-wsize+wsize*j,memw(a-wsize+wsize*j))
        end;
    CSA: begin k:=argw(k); if k<>wsize then trap(EILLINS);
          a:=popa;
          st:= popsw - signwd(memw(a+asize));
          if (st>=0) and (st<=memw(a+wsize+asize)) then
             b:=mema(a+2*wsize+asize+asize*st) else b:=mema(a);
          if b=0 then trap(ECASE) else newpc(b)
        end;
    CSB: begin k:=argw(k); if k<>wsize then trap(EILLINS); a:=popa;
          t:=pop; i:=1; found:=false;
          while (i<=memw(a+asize)) and not found do
            if t=memw(a+(asize+wsize)*i) then found:=true else i:=i+1;
          if found then b:=memw(a+(asize+wsize)*i+wsize) else b:=memw(a);
          if b=0 then trap(ECASE) else newpc(b);
        end;
    DCH: begin pusha(mema(popa+dynd)) end;
    DUP,DUS:
        begin if insr=DUS then
             begin k:=argw(k); if k<>wsize then trap(EILLINS); k:=pop end;
          k:=args(k);
          for i:=1 to k div wsize do push(memw(sp+k-wsize));
        end;
    EXG: begin
          k:=argw(k);
          for i:=1 to k div wsize do push(memw(sp+k-wsize));
          for i:=0 to k div wsize - 1 do
```

```
          store(sp+k+i*wsize,memw(sp+k+k+i*wsize));
        for i:=1 to k div wsize do
        begin t:=pop ; store(sp+k+k-wsize,t) end;
      end;
 FIL: filna(argg(k));
 GTO: begin k:=argg(k);
        newlb(mema(k+2*asize)); newsp(mema(k+asize)); newpc(mema(k))
      end;
 LIM: push(ignmask);
 LIN: lino(argn(k));
 LNI: lino(memw(0)+1);
 LOR: begin i:=argr(k);
        case i of 0:pusha(lb); 1:pusha(sp); 2:pusha(hp) end;
      end;
 LPB: pusha(popa+statd);
 MON: domon(pop);
 NOP: writeln('NOP at line ',memw(0):5) ;
 RCK: begin a:=popa;
        case szindex(argw(k)) of
        1: if (signwd(memw(sp))<signwd(memw(a))) or
              (signwd(memw(sp))>signwd(memw(a+wsize))) then trap(ERANGE);
        2: if (memd(sp)<memd(a)) or
              (memd(sp)>memd(a+2*wsize)) then trap(ERANGE);
        end
      end;
 RTT: dortt;
 SIG: begin a:=popa; pusha(uerrorproc); uerrorproc:=a end;
 SIM: ignmask:=pop;
 STR: begin i:=argr(k);
        case i of 0: newlb(popa); 1: newsp(popa); 2: newhp(popa) end;
      end;
 TRP: trap(pop)
end
end;
```

```
{---------------------------------------------------------------------------}
{                              Main Loop                                    }
{---------------------------------------------------------------------------}

begin initialize;
8888:
  repeat
    opcode := nextpc;       { fetch the first byte of the instruction }
    if opcode=escape1 then iclass:=second
    else if opcode=escape2 then iclass:=tert
    else iclass:=prim;
    if iclass<>prim then opcode := nextpc;
    with dispat[iclass][opcode] do
      begin insr:=instr;
        if not (zbit in iflag) then
          if ibit in iflag then k:=pop else
            begin
              if mini in iflag then k:=implicit else
                begin
                  if short in iflag then k:=implicit+nextpc else
                    begin k:=nextpc;
                      if (sbit in iflag) and (k>=128) then k:=k-256;
                      for i:=2 to ilength do k:=256*k + nextpc
                    end
                end;
              if wbit in iflag then k:=k*wsize;
            end
      end;
case insr of

  NON: trap(EILLINS);

  { LOAD GROUP }
  LDC,LOC,LOL,LOE,LIL,LOF,LAL,LAE,LXL,LXA,LOI,LOS,LDL,LDE,LDF,LPI:
      loadops;

  { STORE GROUP }
  STL,STE,SIL,STF,STI,STS,SDL,SDE,SDF:
      storeops;

  { SIGNED INTEGER ARITHMETIC }
  ADI,SBI,MLI,DVI,RMI,NGI,SLI,SRI:
      intarith;

  { UNSIGNED INTEGER ARITHMETIC }
  ADU,SBU,MLU,DVU,RMU,SLU,SRU:
      unsarith;

  { FLOATING POINT ARITHMETIC }
  ADF,SBF,MLF,DVF,NGF,FIF,FEF:
      fltarith;

  { POINTER ARITHMETIC }
  ADP,ADS,SBS:
      ptrarith;

  { INCREMENT/DECREMENT/ZERO }
  INC,INL,INE,DEC,DEL,DEE,ZRL,ZRE,ZER,ZRF:
      incops;
```

```
  { CONVERT GROUP }
  CII,CIU,CIF,CUI,CUU,CUF,CFI,CFU,CFF:
      convops;

  { LOGICAL GROUP }
 XAND,IOR,XOR,COM,ROL,ROR:
      logops;

  { SET GROUP }
  INN,XSET:
      setops;

  { ARRAY GROUP }
  LAR,SAR,AAR:
      arrops;

  { COMPARE GROUP }
  CMI,CMU,CMP,CMF,CMS,  TLT,TLE,TEQ,TNE,TGE,TGT:
      cmpops;

  { BRANCH GROUP }
  BRA,  BLT,BLE,BEQ,BNE,BGE,BGT,  ZLT,ZLE,ZEQ,ZNE,ZGE,ZGT:
      branchops;

  { PROCEDURE CALL GROUP }
  CAL,CAI,RET,LFR:
      callops;

  { MISCELLANEOUS GROUP }
  ASP,ASS,BLM,BLS,CSA,CSB,DCH,DUP,DUS,EXG,FIL,GTO,LIM,
  LIN,LNI,LOR,LPB,MON,NOP,RCK,RTT,SIG,SIM,STR,TRP:
      miscops;

    end;        { end of case statement }
    if not ( (insr=RET) or (insr=ASP) or (insr=BRA) or (insr=GTO) ) then
        retsize:=0 ;
  until halted;
9999:
  writeln('halt with exit status: ',exitstatus:1);
  doident;
end.
```

B. EM CODE TABLES

The following table is used by the assembler for EM machine language. It specifies the opcodes used for each instruction and how arguments are mapped to machine language arguments. The table is presented in three columns, each line in each column contains three or four fields. Each line describes a range of interpreter opcodes by specifying for which instruction the range is used, the type of the opcodes (mini, shortie, etc..) and range for the instruction argument.

The first field on each line gives the EM instruction mnemonic, the second field gives some flags. If the opcodes are minis or shorties the third field specifies how many minis/shorties are used. The last field gives the number of the (first) interpreter opcode.

Flags :

Opcode type, only one of the following may be specified.

− opcode without argument

m mini

s shortie

2 opcode with 2-byte signed argument

4 opcode with 4-byte signed argument

8 opcode with 8-byte signed argument

u opcode with 2-byte unsigned argument

Secondary (escaped) opcodes.

e The opcode thus marked is in the secondary opcode group instead of the primary

restrictions on arguments

N Negative arguments only

P Positive and zero arguments only

mapping of arguments

w argument must be divisible by the wordsize and is divided by the wordsize before use as opcode argument.

o argument (possibly after division) must be >= 1 and is decremented before use as opcode argument

If the opcode type is 2,4 or 8 the resulting argument is used as opcode argument (least significant byte first).
If the opcode type is mini, the argument is added to the first opcode − if in range − . If the argument is negative, the absolute value minus one is used in the algorithm above.
For shorties with positive arguments the first opcode is used for arguments in the range 0..255, the second for the range 256..511, etc.. For shorties with negative arguments the first opcode is used for arguments in the range −1..−256, the second for the range −257..−512, etc.. The byte following the opcode contains the least significant byte of the argument. First some examples of these specifications.

aar mwPo 1 34
: Indicates that opcode 34 is used as a mini for Positive instruction arguments only. The w and o indicate division and decrementing of the instruction argument. Because the resulting argument must be zero (only opcode 34 may be used), this mini can only be used for instruction argument 2. Conclusion: opcode 34 is for "AAR 2".

adp sP 1 41
Opcode 41 is used as shortie for ADP with arguments in the range 0..255.

bra sN 2 60
Opcode 60 is used as shortie for BRA with arguments −1..−256, 61 is used for arguments −257..−512.

zer e− 145
Escaped opcode 145 is used for ZER.

The interpreter opcode table:

aar	mwPo	1	34	adf	sP	1	35	adi	mwPo	2	36
adp	2		38	adp	mPo	2	39	adp	sP	1	41
adp	sN	1	42	ads	mwPo	1	43	and	mwPo	1	44
asp	mwPo	5	45	asp	swP	1	50	beq	2		51
beq	sP	1	52	bge	sP	1	53	bgt	sP	1	54
ble	sP	1	55	blm	sP	1	56	blt	sP	1	57
bne	sP	1	58	bra	2		59	bra	sN	2	60
bra	sP	2	62	cal	mPo	28	64	cal	sP	1	92
cff	−		93	cif	−		94	cii	−		95
cmf	sP	1	96	cmi	mwPo	2	97	cmp	−		99
cms	sP	1	100	csa	mwPo	1	101	csb	mwPo	1	102
dec	−		103	dee	sw	1	104	del	swN	1	105
dup	mwPo	1	106	dvf	sP	1	107	dvi	mwPo	1	108
fil	u		109	inc	−		110	ine	w2		111
ine	sw	1	112	inl	mwN	3	113	inl	swN	1	116
inn	sP	1	117	ior	mwPo	1	118	ior	sP	1	119
lae	u		120	lae	sw	7	121	lal	P2		128
lal	N2		129	lal	mP	1	130	lal	mN	1	131
lal	swP	1	132	lal	swN	2	133	lar	mwPo	1	135
ldc	mP	1	136	lde	w2		137	lde	sw	1	138
ldl	mP	1	139	ldl	swN	1	140	lfr	mwPo	2	141
lfr	sP	1	143	lil	swN	1	144	lil	swP	1	145
lil	mwP	2	146	lin	2		148	lin	sP	1	149
lni	−		150	loc	2		151	loc	mP	34	0
loc	mN	1	152	loc	sP	1	153	loc	sN	1	154
loe	w2		155	loe	sw	5	156	lof	2		161
lof	mwPo	4	162	lof	sP	1	166	loi	2		167
loi	mPo	1	168	loi	mwPo	4	169	loi	sP	1	173
lol	wP2		174	lol	wN2		175	lol	mwP	4	176
lol	mwN	8	180	lol	swP	1	188	lol	swN	1	189
lxa	mPo	1	190	lxl	mPo	2	191	mlf	sP	1	193
mli	mwPo	2	194	rck	mwPo	1	196	ret	mwP	2	197
ret	sP	1	199	rmi	mwPo	1	200	sar	mwPo	1	201
sbf	sP	1	202	sbi	mwPo	2	203	sdl	swN	1	205
set	sP	1	206	sil	swN	1	207	sil	swP	1	208
sli	mwPo	1	209	ste	w2		210	ste	sw	3	211
stf	2		214	stf	mwPo	2	215	stf	sP	1	217
sti	mPo	1	218	sti	mwPo	4	219	sti	sP	1	223
stl	wP2		224	stl	wN2		225	stl	mwP	2	226
stl	mwN	5	228	stl	swN	1	233	teq	−		234
tgt	−		235	tlt	−		236	tne	−		237
zeq	2		238	zeq	sP	2	239	zer	sP	1	241
zge	sP	1	242	zgt	sP	1	243	zle	sP	1	244
zlt	sP	1	245	zne	sP	1	246	zne	sN	1	247
zre	w2		248	zre	sw	1	249	zrl	mwN	2	250
zrl	swN	1	252	zrl	wN2		253	aar	e2		0
aar	e−		1	adf	e2		2	adf	e−		3
adi	e2		4	adi	e−		5	ads	e2		6
ads	e−		7	adu	e2		8	adu	e−		9

and	e2		10	and	e−		11	asp	ew2		12
ass	e2		13	ass	e−		14	bge	e2		15
bgt	e2		16	ble	e2		17	blm	e2		18
bls	e2		19	bls	e−		20	blt	e2		21
bne	e2		22	cai	e−		23	cal	e2		24
cfi	e−		25	cfu	e−		26	ciu	e−		27
cmf	e2		28	cmf	e−		29	cmi	e2		30
cmi	e−		31	cms	e2		32	cms	e−		33
cmu	e2		34	cmu	e−		35	com	e2		36
com	e−		37	csa	e2		38	csa	e−		39
csb	e2		40	csb	e−		41	cuf	e−		42
cui	e−		43	cuu	e−		44	dee	ew2		45
del	ewP2		46	del	ewN2		47	dup	e2		48
dus	e2		49	dus	e−		50	dvf	e2		51
dvf	e−		52	dvi	e2		53	dvi	e−		54
dvu	e2		55	dvu	e−		56	fef	e2		57
fef	e−		58	fif	e2		59	fif	e−		60
inl	ewP2		61	inl	ewN2		62	inn	e2		63
inn	e−		64	ior	e2		65	ior	e−		66
lar	e2		67	lar	e−		68	ldc	e2		69
ldf	e2		70	ldl	ewP2		71	ldl	ewN2		72
lfr	e2		73	lil	ewP2		74	lil	ewN2		75
lim	e−		76	los	e2		77	los	e−		78
lor	esP	1	79	lpi	e2		80	lxa	e2		81
lxl	e2		82	mlf	e2		83	mlf	e−		84
mli	e2		85	mli	e−		86	mlu	e2		87
mlu	e−		88	mon	e−		89	ngf	e2		90
ngf	e−		91	ngi	e2		92	ngi	e−		93
nop	e−		94	rck	e2		95	rck	e−		96
ret	e2		97	rmi	e2		98	rmi	e−		99
rmu	e2		100	rmu	e−		101	rol	e2		102
rol	e−		103	ror	e2		104	ror	e−		105
rtt	e−		106	sar	e2		107	sar	e−		108
sbf	e2		109	sbf	e−		110	sbi	e2		111
sbi	e−		112	sbs	e2		113	sbs	e−		114
sbu	e2		115	sbu	e−		116	sde	eu		117
sdf	e2		118	sdl	ewP2		119	sdl	ewN2		120
set	e2		121	set	e−		122	sig	e−		123
sil	ewP2		124	sil	ewN2		125	sim	e−		126
sli	e2		127	sli	e−		128	slu	e2		129
slu	e−		130	sri	e2		131	sri	e−		132
sru	e2		133	sru	e−		134	sti	e2		135
sts	e2		136	sts	e−		137	str	esP	1	138
tge	e−		139	tle	e−		140	trp	e−		141
xor	e2		142	xor	e−		143	zer	e2		144
zer	e−		145	zge	e2		146	zgt	e2		147
zle	e2		148	zlt	e2		149	zne	e2		150
zrf	e2		151	zrf	e−		152	zrl	ewP2		153
dch	e−		154	exg	esP	1	155	exg	e2		156
exg	e−		157	lpb	e−		158	gto	eu		159
ldc	4		0	lae	4		1	lal	P4		2
lal	N4		3	lde	w4		4	ldf	4		5
ldl	wP4		6	ldl	wN4		7	lil	wP4		8
lil	wN4		9	loc	4		10	loe	w4		11
lof	4		12	lol	wP4		13	lol	wN4		14
lpi	4		15	adp	4		16	asp	w4		17
beq	4		18	bge	4		19	bgt	4		20
ble	4		21	blm	4		22	blt	4		23
bne	4		24	bra	4		25	cal	4		26

dee	w4	27	del	wP4	28	del	wN4	29
fil	4	30	gto	4	31	ine	w4	32
inl	wP4	33	inl	wN4	34	lin	4	35
sde	4	36	sdf	4	37	sdl	wP4	38
sdl	wN4	39	sil	wP4	40	sil	wN4	41
ste	w4	42	stf	4	43	stl	wP4	44
stl	wN4	45	zeq	4	46	zge	4	47
zgt	4	48	zle	4	49	zlt	4	50
zne	4	51	zre	w4	52	zrl	wP4	53
zrl	wN4	54	loi	4	55	sti	4	56

The table above results in the following dispatch tables. Dispatch tables are used by interpreters to jump to the routines implementing the EM instructions, indexed by the next opcode. Each line of the dispatch tables gives the routine names of eight consecutive opcodes, preceded by the first opcode number on that line. Routine names consist of an EM mnemonic followed by a suffix. The suffices show the encoding used for each opcode. The following suffices exist:

.z	no arguments
.l	16-bit argument
.L	32-bit argument
.u	16-bit unsigned argument
.lw	16-bit argument divided by the wordsize
.Lw	32-bit argument divided by the wordsize
.p	positive 16-bit argument
.P	positive 32-bit argument
.pw	positive 16-bit argument divided by the wordsize
.Pw	positive 32-bit argument divided by the wordsize
.n	negative 16-bit argument
.N	negative 32-bit argument
.nw	negative 16-bit argument divided by the wordsize
.Nw	negative 32-bit argument divided by the wordsize
.s<num>	shortie with <num> as high order argument byte
.w<num>	shortie with argument divided by the wordsize
.<num>	mini with <num> as argument
.<num>W	mini with <num>*wordsize as argument

<num> is a possibly negative integer.

The dispatch table for the 256 primary opcodes:

0	loc.0	loc.1	loc.2	loc.3	loc.4	loc.5	loc.6	loc.7
8	loc.8	loc.9	loc.10	loc.11	loc.12	loc.13	loc.14	loc.15
16	loc.16	loc.17	loc.18	loc.19	loc.20	loc.21	loc.22	loc.23
24	loc.24	loc.25	loc.26	loc.27	loc.28	loc.29	loc.30	loc.31
32	loc.32	loc.33	aar.1W	adf.s0	adi.1W	adi.2W	adp.l	adp.1
40	adp.2	adp.s0	adp.s−1	ads.1W	and.1W	asp.1W	asp.2W	asp.3W
48	asp.4W	asp.5W	asp.w0	beq.l	beq.s0	bge.s0	bgt.s0	ble.s0
56	blm.s0	blt.s0	bne.s0	bra.l	bra.s−1	bra.s−2	bra.s0	bra.s1
64	cal.1	cal.2	cal.3	cal.4	cal.5	cal.6	cal.7	cal.8
72	cal.9	cal.10	cal.11	cal.12	cal.13	cal.14	cal.15	cal.16
80	cal.17	cal.18	cal.19	cal.20	cal.21	cal.22	cal.23	cal.24
88	cal.25	cal.26	cal.27	cal.28	cal.s0	cff.z	cif.z	cii.z
96	cmf.s0	cmi.1W	cmi.2W	cmp.z	cms.s0	csa.1W	csb.1W	dec.z
104	dee.w0	del.w−1	dup.1W	dvf.s0	dvi.1W	fil.u	inc.z	ine.lw
112	ine.w0	inl.−1W	inl.−2W	inl.−3W	inl.w−1	inn.s0	ior.1W	ior.s0
120	lae.u	lae.w0	lae.w1	lae.w2	lae.w3	lae.w4	lae.w5	lae.w6
128	lal.p	lal.n	lal.0	lal.−1	lal.w0	lal.w−1	lal.w−2	lar.1W
136	ldc.0	lde.lw	lde.w0	ldl.0	ldl.w−1	lfr.1W	lfr.2W	lfr.s0
144	lil.w−1	lil.w0	lil.0W	lil.1W	lin.l	lin.s0	lni.z	loc.l
152	loc.−1	loc.s0	loc.s−1	loe.lw	loe.w0	loe.w1	loe.w2	loe.w3

160	loe.w4	lof.l	lof.1W	lof.2W	lof.3W	lof.4W	lof.s0	loi.l
168	loi.1	loi.1W	loi.2W	loi.3W	loi.4W	loi.s0	lol.pw	lol.nw
176	lol.0W	lol.1W	lol.2W	lol.3W	lol.−1W	lol.−2W	lol.−3W	lol.−4W
184	lol.−5W	lol.−6W	lol.−7W	lol.−8W	lol.w0	lol.w−1	lxa.1	lxl.1
192	lxl.2	mlf.s0	mli.1W	mli.2W	rck.1W	ret.0W	ret.1W	ret.s0
200	rmi.1W	sar.1W	sbf.s0	sbi.1W	sbi.2W	sdl.w−1	set.s0	sil.w−1
208	sil.w0	sli.1W	ste.lw	ste.w0	ste.w1	ste.w2	stf.l	stf.1W
216	stf.2W	stf.s0	sti.1	sti.1W	sti.2W	sti.3W	sti.4W	sti.s0
224	stl.pw	stl.nw	stl.0W	stl.1W	stl.−1W	stl.−2W	stl.−3W	stl.−4W
232	stl.−5W	stl.w−1	teq.z	tgt.z	tlt.z	tne.z	zeq.l	zeq.s0
240	zeq.s1	zer.s0	zge.s0	zgt.s0	zle.s0	zlt.s0	zne.s0	zne.s−1
248	zre.lw	zre.w0	zrl.−1W	zrl.−2W	zrl.w−1	zrl.nw	escape1	escape2

The list of secondary opcodes (escape1):

0	aar.l	aar.z	adf.l	adf.z	adi.l	adi.z	ads.l	ads.z
8	adu.l	adu.z	and.l	and.z	asp.lw	ass.l	ass.z	bge.l
16	bgt.l	ble.l	blm.l	bls.l	bls.z	blt.l	bne.l	cai.z
24	cal.l	cfi.z	cfu.z	ciu.z	cmf.l	cmf.z	cmi.l	cmi.z
32	cms.l	cms.z	cmu.l	cmu.z	com.l	com.z	csa.l	csa.z
40	csb.l	csb.z	cuf.z	cui.z	cuu.z	dee.lw	del.pw	del.nw
48	dup.l	dus.l	dus.z	dvf.l	dvf.z	dvi.l	dvi.z	dvu.l
56	dvu.z	fef.l	fef.z	fif.l	fif.z	inl.pw	inl.nw	inn.l
64	inn.z	ior.l	ior.z	lar.l	lar.z	ldc.l	ldf.l	ldl.pw
72	ldl.nw	lfr.l	lil.pw	lil.nw	lim.z	los.l	los.z	lor.s0
80	lpi.l	lxa.l	lxl.l	mlf.l	mlf.z	mli.l	mli.z	mlu.l
88	mlu.z	mon.z	ngf.l	ngf.z	ngi.l	ngi.z	nop.z	rck.l
96	rck.z	ret.l	rmi.l	rmi.z	rmu.l	rmu.z	rol.l	rol.z
104	ror.l	ror.z	rtt.z	sar.l	sar.z	sbf.l	sbf.z	sbi.l
112	sbi.z	sbs.l	sbs.z	sbu.l	sbu.z	sde.u	sdf.l	sdl.pw
120	sdl.nw	set.l	set.z	sig.z	sil.pw	sil.nw	sim.z	sli.l
128	sli.z	slu.l	slu.z	sri.l	sri.z	sru.l	sru.z	sti.l
136	sts.l	sts.z	str.s0	tge.z	tle.z	trp.z	xor.l	xor.z
144	zer.l	zer.z	zge.l	zgt.l	zle.l	zlt.l	zne.l	zrf.l
152	zrf.z	zrl.pw	dch.z	exg.s0	exg.l	exg.z	lpb.z	gto.u

Finally, the list of opcodes with four byte arguments (escape2).

0	ldc.L	lae.L	lal.P	lal.N	lde.Lw	ldf.L	ldl.Pw	ldl.Nw
8	lil.Pw	lil.Nw	loc.L	loe.Lw	lof.L	lol.Pw	lol.Nw	lpi.L
16	adp.L	asp.Lw	beq.L	bge.L	bgt.L	ble.L	blm.L	blt.L
24	bne.L	bra.L	cal.L	dee.Lw	del.Pw	del.Nw	fil.L	gto.L
32	ine.Lw	inl.Pw	inl.Nw	lin.L	sde.L	sdf.L	sdl.Pw	sdl.Nw
40	sil.Pw	sil.Nw	ste.Lw	stf.L	stl.Pw	stl.Nw	zeq.L	zge.L
48	zgt.L	zle.L	zlt.L	zne.L	zre.Lw	zrl.Pw	zrl.Nw	loi.L
56	sti.L							

C. AN EXAMPLE PROGRAM

```
program example(output);
{This program just demonstrates typical EM code.}
type rec = record r1: integer; r2:real; r3: boolean end;
var mi: integer;  mx:real;  r:rec;

function sum(a,b:integer):integer;
begin
        sum := a + b
end;

procedure test(var r: rec);
label 1;
var i,j: integer;
        x,y: real;
        b: boolean;
        c: char;
        a: array[1..100] of integer;

begin
        j := 1;
        i := 3 * j + 6;
        x := 4.8;
        y := x/0.5;
        b := true;
        c := ’z’;
        for i:= 1 to 100 do a[i] := i * i;
        r.r1 := j+27;
        r.r3 := b;
        r.r2 := x+y;
        i := sum(r.r1, a[j]);
        while i > 0 do begin j := j + r.r1; i := i - 1 end;
        with r do begin r3 := b;  r2 := x+y;  r1 := 0 end;
        goto 1;
1:      writeln(j, i:6, x:9:3, b)
end; {test}
begin {main program}
        mx := 15.96;
        mi := 99;
        test(r)
end.
```

The EM code as produced by the Pascal-VU compiler is given below. Comments have been added manually. Note that this code has already been optimized.

```
 mes 2,2,2                    ; wordsize 2, pointersize 2
.1
 rom 't.p\000'                ; the name of the source file
 hol 552,−32768,0             ; externals and buf occupy 552 bytes
 exp $sum                     ; sum can be called from other modules
 pro $sum,2                   ; procedure sum; 2 bytes local storage
 lin 8                        ; code from source line 8
 ldl 0                        ; load two locals ( a and b )
 adi 2                        ; add them
 ret 2                        ; return the result
 end 2                        ; end of procedure ( still two bytes local storage )
.2
 rom 1,99,2                   ; descriptor of array a[]
 exp $test                    ; the compiler exports all level 0 procedures
 pro $test,226                ; procedure test, 226 bytes local storage
.3
 rom 4.8F8                    ; assemble Floating point 4.8 (8 bytes) in
.4                            ; global storage
 rom 0.5F8                    ; same for 0.5
 mes 3,−226,2,2               ; compiler temporary not referenced by address
 mes 3,−24,2,0                ; the same is true for i, j, b and c in test
 mes 3,−22,2,0
 mes 3,−4,2,0
 mes 3,−2,2,0
 mes 3,−20,8,0                ; and for x and y
 mes 3,−12,8,0
 lin 20                       ; maintain source line number
 loc 1
 stl −4                       ; j := 1
 lni                          ; lin 21 prior to optimization
 lol −4
 loc 3
 mli 2
 loc 6
 adi 2
 stl −2                       ; i := 3 * j + 6
 lni                          ; lin 22 prior to optimization
 lae .3
 loi 8
 lal −12
 sti 8                        ; x := 4.8
 lni                          ; lin 23 prior to optimization
 lal −12
 loi 8
 lae .4
 loi 8
 dvf 8
 lal −20
 sti 8                        ; y := x / 0.5
 lni                          ; lin 24 prior to optimization
 loc 1
 stl −22                      ; b := true
 lni                          ; lin 25 prior to optimization
 loc 122
 stl −24                      ; c := 'z'
 lni                          ; lin 26 prior to optimization
```

```
 loc 1
 stl −2                    ; for i:= 1
2
 lol −2
 dup 2
 mli 2                     ; i*i
 lal −224
 lol −2
 lae .2
 sar 2                     ; a[i] :=
 lol −2
 loc 100
 beq *3                    ; to 100 do
 inl −2                    ; increment i and loop
 bra *2
3
 lin 27
 lol −4
 loc 27
 adi 2                     ; j + 27
 sil 0                     ; r.r1 :=
 lni                       ; lin 28 prior to optimization
 lol −22                   ; b
 lol 0
 stf 10                    ; r.r3 :=
 lni                       ; lin 29 prior to optimization
 lal −20
 loi 16
 adf 8                     ; x + y
 lol 0
 adp 2
 sti 8                     ; r.r2 :=
 lni                       ; lin 30 prior to optimization
 lal −224
 lol −4
 lae .2
 lar 2                     ; a[j]
 lil 0                     ; r.r1
 cal $sum                  ; call now
 asp 4                     ; remove parameters from stack
 lfr 2                     ; get function result
 stl −2                    ; i :=
4
 lin 31
 lol −2
 zle *5                    ; while i > 0 do
 lol −4
 lil 0
 adi 2
 stl −4                    ; j := j + r.r1
 del −2                    ; i := i - 1
 bra *4                    ; loop
5
 lin 32
 lol 0
 stl −226                  ; make copy of address of r
 lol −22
 lol −226
 stf 10                    ; r3 := b
```

```
 lal −20
 loi 16
 adf 8
 lol −226
 adp 2
 sti 8                  ; r2 := x + y
 loc 0
 sil −226               ; r1 := 0
 lin 34                 ; note the absence of the unnecessary jump
 lae 22                 ; address of output structure
 lol −4
 cal $_wri              ; write integer with default width
 asp 4                  ; pop parameters
 lae 22
 lol −2
 loc 6
 cal $_wsi              ; write integer width 6
 asp 6
 lae 22
 lal −12
 loi 8
 loc 9
 loc 3
 cal $_wrf              ; write fixed format real, width 9, precision 3
 asp 14
 lae 22
 lol −22
 cal $_wrb              ; write boolean, default width
 asp 4
 lae 22
 cal $_wln              ; writeln
 asp 2
 ret 0                  ; return, no result
 end 226
 exp $_main
 pro $_main,0           ; main program
.6
 con 2,−1,22            ; description of external files
.5
 rom 15.96F8
 fil .1                 ; maintain source file name
 lae .6                 ; description of external files
 lae 0                  ; base of hol area to relocate buffer addresses
 cal $_ini              ; initialize files, etc...
 asp 4
 lin 37
 lae .5
 loi 8
 lae 2
 sti 8                  ; mx := 15.96
 lni                    ; lin 38 prior to optimization
 loc 99
 ste 0                  ; mi := 99
 lni                    ; lin 39 prior to optimization
 lae 10                 ; address of r
 cal $test
 asp 2
 loc 0                  ; normal exit
 cal $_hlt              ; cleanup and finish
```

```
asp 2
end 0
mes 5                        ; reals were used
```

The compact code corresponding to the above program is listed below. Read it horizontally, line by line, not column by column. Each number represents a byte of compact code, printed in decimal. The first two bytes form the magic word.

```
173   0 159 122 122 122 255 242   1 161 250 124 116  46 112   0
255 156 245  40   2 245   0 128 120 155 249 123 115 117 109 160
249 123 115 117 109 122  67 128  63 120   3 122  88 122 152 122
242   2 161 121 219 122 255 155 249 124 116 101 115 116 160 249
124 116 101 115 116 245 226   0 242   3 161 253 128 123  52  46
 56 255 242   4 161 253 128 123  48  46  53 255 159 123 245  30
255 122 122 255 159 123  96 122 120 255 159 123  98 122 120 255
159 123 116 122 120 255 159 123 118 122 120 255 159 123 100 128
120 255 159 123 108 128 120 255  67 140  69 121 113 116  68  73
116  69 123  81 122  69 126   3 122 113 118  68  57 242   3  72
128  58 108 112 128  68  58 108  72 128  57 242   4  72 128  44
128  58 100 112 128  68  69 121 113  98  68  69 245 122   0 113
 96  68  69 121 113 118 182  73 118  42 122  81 122  58 245  32
255  73 118  57 242   2  94 122  73 118  69 220  10 123  54 118
 18 122 183  67 147  73 116  69 147   3 122 104 120  68  73  98
 73 120 111 130  68  58 100  72 136   2 128  73 120   4 122 112
128  68  58 245  32 255  73 116  57 242   2  59 122  65 120  20
249 123 115 117 109   8 124  64 122 113 118 184  67 151  73 118
128 125  73 116  65 120   3 122 113 116  41 118  18 124 185  67
152  73 120 113 245  30 255  73  98  73 245  30 255 111 130  58
100  72 136   2 128  73 245  30 255   4 122 112 128  69 120 104
245  30 255  67 154  57 142  73 116  20 249 124  95 119 114 105
  8 124  57 142  73 118  69 126  20 249 124  95 119 115 105   8
126  57 142  58 108  72 128  69 129  69 123  20 249 124  95 119
114 102   8 134  57 142  73  98  20 249 124  95 119 114  98   8
124  57 142  20 249 124  95 119 108 110   8 122  88 120 152 245
226   0 155 249 125  95 109  97 105 110 160 249 125  95 109  97
105 110 120 242   6 151 122 119 142 255 242   5 161 253 128 125
 49  53  46  57  54 255  50 242   1  57 242   6  57 120  20 249
124  95 105 110 105   8 124  67 157  57 242   5  72 128  57 122
112 128  68  69 219 110 120  68  57 130  20 249 124 116 101 115
116   8 122  69 120  20 249 124  95 104 108 116   8 122 152 120
159 124 160 255 159 125 255
```