Andrew S. Tanenbaum, Johan W. Stevenson and H. van Staveren

Description of an Experimental machine Architecture for use with Block Structured Languages

Rapport nr. IR-54

VRIJE UNIVERSITEIT AMSTERDAM

Wiskundig Seminarium, De Boelelaan 1081, 1081 HV Amsterdam

DESCRIPTION OF AN EXPERIMENTAL MACHINE ARCHITECTURE FOR USE WITH BLOCK STRUCTURED LANGUAGES

Andrew S. Tanenbaum
Johan W. Stevenson
Hans van Staveren

April 1980

Informatica Rapport IR-54

Abstract

EM-1 is a machine architecture designed to be interpreted on microcomputers. It has an instruction set convenient for compilers as well as a compact object program format. Furthermore, the encoding has been done to make interpretation with a cost penalty of roughly a factor of 5-8 feasible. This document describes the machine architecture, its instructions and their meanings.

TABLE OF CONTENTS

1. INTRODUCTION

EM-1 is an Experimental Machine architecture designed with the following goals in mind:

1. A compact instruction set, to reduce the amount of memory needed for program storage, and to reduce the time needed to transmit programs over communication lines.

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

3. It should be designed with microprogrammed implementations in mind; in particular, the use of many short fields within instruction opcodes should be avoided, since their extraction by the microprogram is inefficient.

4. The design should allow interpretation on, or translation to, 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.

5. It should be relatively easy to adapt the machine to new technological trends such as 32-bit address spaces or 32-bit machine words.

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 a small number of built-in object types, for example single and double precision 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 two objects from the top of the stack, perform an operation on them, and push the result back onto the stack.

There are no visible general registers used for arithmetic operands etc. This is in contrast to most third generation computerw, 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 small number of internal registers with specific functions as follows:

PC: Program Counter. Byte number of next instruction
LB: Local Base. Points to zeroth local variable in the current procedure.
SP: Stack Pointer. Points to the highest occupied word on the stack.
HP: Heap Pointer. Points to the bottom 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. High performance can be achieved 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-1 instructions have zero or one operand. We believe that instructions needing two operands 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 the EM-1 machine 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-1 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 levels 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 pseudo instructions. The machine language is designed to make program text compact and to make decoding easy. The binary representation of the machine 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. The encoding is nevertheless efficient: within 5% of the optimal Huffman code, measured over a large sample of Pascal programs.

A common use for EM-1 is for producing portable (cross) compilers. When used this way, the compilers produce EM-1 assembly language as their output. To run the compiled program on the target machine, another program, called 'back end', translates the EM-1 assembly language to the target machine's assembler. When this approach is used, the format of the EM-1 machine language instructions is irrelevant. On the other hand, when writing an interpreter for EM-1 machine language programs, the interpreter must deal with the machine language and not with the assembly language. This point is not trivial, because the semantics of many EM-1 assembly language instructions systematically differ from their machine language counterparts. For example, if an assembly language operand must be even, the machine language representation may not include the low order bit, which is always zero.

The machine described in this document, EM-1, uses machine words of 16 bits and pointers fitting in one machine word. There are two planned successors. One, called EM-2, uses two word pointers to address very large address spaces. The other successor, EM-3, has machine words of 32 bits. A single word is used for pointers in EM-3.

2. MEMORY

The EM-1 machine has two distinct address spaces, one for instruction and one for data. Each address space is divided up into 8-bit bytes. Bytes are the smallest addressable units. Bytes are numbered consecutively from 0 to some maximum. For the EM-1 machine we have made the following choices:

DECISION A-1: The instruction address space consists of up to 2 ** 16 bytes.
DECISION B-1: The data address space consists of up to 2 ** 16 bytes.

Therefore, all addresses can be represented in two bytes.

While bytes are the smallest addressable units, the basic unit in EM-1 is the machine word. The low order byte of a machine word is the rightmost byte, and has an address that is a multiple of the wordsize n. Figure 1 shows the first two words, with addresses 0 and n.

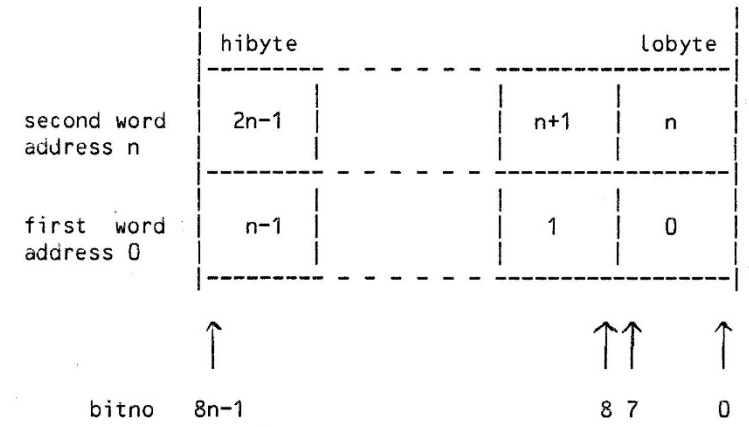


Fig. 1. byte-word addressing

The first word contains bytes 0,1,...,n-1; the second word contains bytes n,n+1,...,2n-1 etc. Machine words are addressed by the address of the low order byte. For the EM-1 machine we have made the following choice:

DECISION C-1: A machine word consists of two bytes.

Thus all EM-1 words have even addresses.

The size of all objects in EM-1 is an integral number of words with one exception: objects consisting of a single byte. There are only two operations allowed on single byte objects: push it onto the stack and pop it from the stack. The addressing of bytes in memory is always indirect. If a single byte object is pushed onto the stack it is put in the low order byte of a word and the rest of the word is cleared to zero. Popping a byte from the stack removes a word from the stack, stores the low order byte of this word in memory and discards the rest of the word.

The assembly language format of instructions that reference memory by offset, requires the offset to be a multiple of the word size. For example, LOL 8 means fetch the local variable 8 bytes from the base of the current stack frame. To make compact encoding possible the offset actually present in the machine instructions is given in words, rather than in bytes. However, for simplicity, the offset specified in the assembly language is always in bytes. The compiler writer and the back end writer need not be aware of the details of the machine encoding.

The same remarks as for offsets apply to the size of objects. Sizes in assembly language are always in bytes and are one byte or a multiple of the wordsize. They are specified in words in the machine language and there is a special encoding for the size of one byte.

Some instructions fetch their offset or object size argument indirectly, for instance from the top of the stack. These offsets and sizes are always in bytes, so that the instruction itself must check that the addresses are word addresses or that the sizes are a single byte or word multiples.

The format of pointers into both address spaces is explicitly undefined. Each implementor of EM-1 may choose a suitable representation of pointers himself. The size of a pointer, however, is fixed, so that the compiler writer knows how much storage to allocate for a pointer. For EM-1 the size of pointers into the data address space as well as into the instruction address space is a single word.

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 single precision integer value 0 as nil pointer. This value is also used by many C programs as 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 since the first word in the EM-1 data space is special and probably not the target of any pointer.

The next two chapters describe the EM-1 memory in more detail. One describes the instruction address space, the other the data address space. Figure 2 gives an overview of these memory partitions.

A design goal of EM-1 has been to allow its implementation on a wide range of existing machines, not only designing a new one to be built in hardware. To this extent we have tried to minimize the demands of EM-1 on the memory structure of the target machine. Therefore, apart from the logical partitioning, EM-1 memory is divided into 'fragments'. A fragment consists of consecutive machine words and has a base address and a size. The only way to address the words of a fragment is by offsetting from its base address. The base address may only be used to address words of the corresponding fragment, i.e. offsets greater than the fragment's size are illegal.

It will be clear that fragments may be allocated anywhere in the address space and that only the calculation of the base address must be adapted. Reallocation of fragments at run time, however, is not trivial, because pointers to fragment words may be stored, which must then be relocated.

While following the logical partitioning in the next chapters, we will indicate the relation to fragments.

```
1.              Instruction address space
1.1.                  Procedure descriptor table
1.1.1.                      Number of bytes for parameters
1.1.2.                      Procedure base address
1.1.x                       ......
1.2.                  Program text
1.2.1.                      Procedure text
2.              Data address space
2.1.                  Global data area
2.1.1.                      Global data blocks
2.1.1.1.                        ABS block
2.1.1.1.1.                          Line number
2.1.1.1.x.                          ..........
2.1.1.2.                        CON blocks
2.1.1.3.                        ROM blocks
2.1.1.4.                        BSS blocks
2.1.1.5.                        HOL blocks
2.1.2.                      Local data area
2.1.2.1.                        Procedure frame
2.1.2.1.1.                          Mark block zone
2.1.2.1.1.1.                            Static link
2.1.2.1.1.2.                            Dynamic link
2.1.2.1.1.3.                            Return address
2.1.2.1.1.x.                            ............
2.1.2.1.2.                          Actual parameter zone
2.1.2.1.3.                          Local variable zone
2.1.2.1.4.                          Dynamic local generator zone
2.1.2.2.                        Stack
2.1.3.                      Heap data area
```

Fig. 2. Memory overview.

3. INSTRUCTION ADDRESS SPACE

The instruction space of the EM-1 machine contains the procedure descriptor table and the binary machine code for procedures. The instruction space does not change during the execution of a program, so that it may be protected.

The procedure descriptor table contains an entry for each procedure in the program. The size of a descriptor is explicitly undefined. A descriptor must contain at least two items:

1. A constant telling how many bytes of parameters the procedure has.
2. The base address of the procedure's code. This is the only place that procedure addresses are used.

A descriptor may contain other (implementation dependent) information for debugging or monitoring. The most important reason to have a procedure descriptor table is that it makes very compact procedure calls possible. It may not be necessary for some implementations, however, to have a procedure descriptor table at all.

Each procedure has a single entry point: the first instruction, located at the base address stored in the descriptor. There is a single EM-1 program counter PC pointing to the next instruction to be executed. The procedure pointed into by PC is called the 'current' procedure. A procedure may call another procedure using the CAL or CAS instruction. The calling procedure remains 'active' and is resumed whenever the called procedure returns. Note that a procedure may have many 'active' invocations.

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 HLT instruction that stops the program, exiting all active procedures.
- the RET or RES instruction, which returns to the calling procedure.
- the RTT instruction, which exits a trap handling routine and resumes the trapping instruction (see below).
- a non-local jump using the CSA or CSB instructions. These instructions are normally used for case statements to jump to selected pieces of code local to the procedure that executes them. If these instructions must be used to exit from procedures you have to proceed with great care, because you have to unravel the stack yourself! This way of exiting a procedure should only be used in emergencies.

All branch instructions are relative to the program counter. There is only one branch instruction that jumps backward: BRB, unconditional branch backward. Branch instructions can never jump out of a procedure. Because the operand of a branch instruction must have a fixed range accommodating all possible offsets, there must be a maximum procedure size:

DECISION D-1: The code for a single procedure must fit in 2 ** 15 bytes.

The program text for each procedure, as well as the descriptor table, 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 unique 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. However, constructing pointers is different for these data areas.

Global data is allocated by using several pseudo instructions in the assembly language, as described below. 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 operand. Examples are LOE, LAE and STE. The operands of these instructions range from 0 to some maximum.

DECISION E-1: The maximum size of global data is 2 ** 15 bytes

Part of the global data area is initialized by the compiler, part of it is initialized with program arguments supplied by the user and the rest is not initialized at all or is initialized with some arbitrary 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 contains the stack and some data for each active procedure invocation, called a 'frame'. The size of the local data area varies. The base address is fixed, so it grows from a low EM-1 address to a high EM-1 address. On top of the current procedure frame resides the operand stack. The stack pointer SP always points to the top of the stack and it thereby marks the high end 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 can be addressed directly. Variables in other active procedures are addressed by following the chain of statically enclosing procedures using the LEX instruction. Many instructions have offsets to LB as argument, for instance LOL, LAL and STL. The argument of these instructions range from 0 to some maximum.

DECISION F-1: The maximum size of a procedure frame including the operand stack is 2 ** 15 bytes.

The procedure call instructions CAL and CAS transform some words on top of the stack into a new frame. Each procedure, therefore, starts with an empty stack. The return instructions RET, RES and RTT remove a frame. RET and RES may copy some words on top of the stack of the returning procedure to the top of the stack of the previous routine, as result value.

The heap data area grows downwards. It is initially empty. The initial value of the heap pointer HP marks the high end. The heap pointer may be manipulated by the LOR and STR instructions. The heap can only be addressed indirectly, by pointers obtained from previous values of HP.

4.1. GLOBAL DATA AREA

The size of global data is fixed at compile time. Global data is allocated by several pseudo instructions in the EM-1 assembly language. This section anticipates the description of the assembly language below, but the concepts are similar to other assembly languages. Each pseudo instruction allocates some words. The words 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 words of these blocks are consecutive.

Initialized global data is allocated by the pseudo instruction CON. It needs at least one operand. There are several operand types: single and double precision integer constants, floating point constants, procedure numbers, instruction labels, addresses in the global data area and strings. Strings are byte sequences. For each operand an integral number of words, determined by the operand type, is allocated and initialized. The assembler pads strings out to an integral number of words.

The pseudo instruction 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 perform certain calculations such as array indexing and subrange checking at compile time instead of at run time.

The pseudo instruction BSS allocates uninitialized global data. The only operand to this pseudo is the number of bytes required, which must be a multiple of the wordsize. All words of a BSS block may have different initial values, but most implementations will assign some fixed value to these words, typically -32768 or 0.

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 global data blocks by prepending CON, ROM and BSS pseudos with alphanumeric labels. Moreover, the only way to address these named global data blocks is by using their 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 pseudo instructions to initialize pointers.

Unlike many other assembly languages, the EM-1 assembly language requires all operands of normal and pseudo instructions to be either a number or an identifier, but not an expression. This makes it impossible to address the third word of a ten word BSS block directly. Thus LOE LABEL+4 is not permitted, nor is CON LABEL+4. To access LABEL+4 you must first load the base address of the BSS block using its name (LABEL). Then a second instruction may add an offset (4) to this address and can load the addressed object onto the stack. This restriction, although annoying, is fully intentional, and greatly aids optimization.

The pseudo instruction HOL offers a partial solution to this problem. HOL is similar to BSS in that it requests an uninitialized global data block with size equal to the value of its operand (in bytes, but a multiple of the wordsize). Addressing of a HOL block, however, is quasi absolute. The first word is addressed by 0, the second word by 2 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 (EOF pseudo) whatever comes first. Each instruction falls in the scope of at most one HOL block, the current HOL block.

A fifth type of global data is a small block, called ABS block, with implementation defined size. It is addressed absolutely, both in assembly language and in the machine 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. Other items in this block might be the arguments supplied by the caller of this program.

Note that all numeric operands of the instructions that address the global data area refer to the current HOL block if they are preceded by at least one HOL pseudo in the same module and that they refer to the ABS block otherwise. Thus LOE 0 loads the zeroth word in the most recent HOL, unless no HOL has appeared in the current file, in which case it loads the zeroth word in the ABS fragment.

The global data area is highly fragmented. The ABS block and each HOL block form a separate fragment. The situation for CON, ROM and BSS blocks is more complex. The assembler groups several blocks into a single fragment. A fragment only contains blocks of the same type: CON, ROM or BSS. It is guaranteed that the words allocated for two CON pseudos are allocated consecutively in a single fragment, unless these CON pseudos are separated in the assembly language program by one or more of the following pseudos:

ROM, BSS, HOL, END and EOF

A similar statement can be made for ROM pseudos and for BSS pseudos.

4.2. LOCAL DATA AREA

The local data area consists of a sequence of frames, one for each active procedure. On top of the frame of the current procedure resides the stack. Frames are generated by procedure calls and are removed by procedure returns. A procedure frame consists of 5 'zones':

1. The mark block.
2. The actual parameters
3. The local variables and compiler temporaries
4. The dynamic local generators
5. The operand stack.

A sample frame is shown in Fig. 3.

The first step in the procedure calling sequence is to deposit a mark block on the stack of the current procedure (using the MRK or MRS instruction). The mark block contains at least the following items:

- static link: the LB value of the most recent invocation of the statically enclosing procedure. This field supports the linkage needed by block structured languages.

- dynamic link: the LB value of the calling procedure.

- return address of the calling procedure, pointing into the instruction space.

The exact format and encoding of the mark block is explicitly undefined and may vary from implementation to implementation, i.e. it is up to the implementor to decide the number of machine words, their order and their encoding.

The second step in the calling sequence is to push the actual parameters on top of the stack of the calling procedure. The exact details are compiler dependent. If a procedure has no parameters this zone will be empty. Note that the evaluation of the actual parameters may imply the calling of procedures.

The third and last step is to call the procedure using the CAL or CAS instruction. Several tasks are performed by the call instructions. First, the LB is changed to point to the first word above the mark block, containing the first parameter if present. The new LB is calculated by subtracting the number of bytes passed as parameter (see procedure descriptor) from the stack pointer SP. Second, the old program counter PC is saved in the mark block, some words below the new LB. Third, the new program counter is fetched from the procedure descriptor and stored in PC.

Normally the first instruction of the called procedure will be BEG to advance SP, reserving space for local variables and compiler temporaries. The initial value of the allocated words is not defined, but implementations that check for undefined values will probably initialize them with the special 'undefined' pattern, typically -32768. This same instruction BEG may also be used to remove some words from the stack.

There is a version of BEG, called BES, 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. By allocating the objects of known size first, these can be addressed using the more efficient instructions with fixed offset.

The distinction between the last four zones of procedure frames is rather vague. The only restriction is that SP may not be smaller than LB and that only words between LB and SP may be accessed by offsetting to LB.

It should be noted that procedures with a variable number of parameters cannot be accommodated. Each such procedure must have some maximum number of bytes worth of parameters, and exactly this number must be passed on each call. Of course the programmer or compiler is free to use the BEG instruction in the calling procedure to advance the stack pointer, thus simulating the passing of many dummy parameters in one instruction.

Each procedure frame is a separate fragment. Since any fragment may be placed anywhere in memory, procedure frames need not be contiguous. If for some reason it is necessary to implement frames non-contiguously, the call in-

struction must copy the mark block and actual parameters to the location of the new frame. However, this movement implies that an actual parameter that points to another actual parameter in the same fragment has to be relocated, an impossible task. Consequently, on implementations that do not store procedure frames contiguously, programs must not allow actual parameters to point to other actual parameters in the same procedure. It is difficult, however, to construct an example for which this restriction is prohibitive.

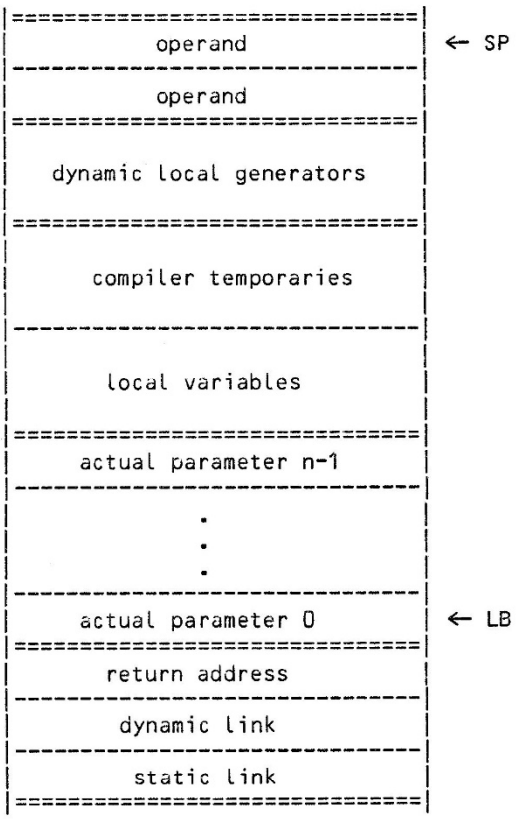


Fig. 3. A sample procedure frame.

4.3. HEAP DATA AREA

The heap area starts empty, with HP pointing to the high end of it. HP always contains a word address. The current value of HP can always be obtained by the LOR instruction. (Note: we use 'value' to indicate the byte number in the data space, independent of the pointer representation). A new value may be

stored in the heap pointer using the STR instruction. If the new value is smaller than the old one, then the heap grows downwards. If it is greater, then the heap shrinks. The value of HP may never exceed the original value of HP. 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 for local or global data. If this is attempted the STR instruction will cause a trap to occur.

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

The heap is a single fragment. All words have consecutive addresses. There are no limits on the size of the heap as long as it fits in the data address space.

5. MAPPING OF EM-1 MEMORY ONTO TARGET MACHINE MEMORY

The EM-1 architecture is designed to be implemented on a large number of existing and future machines. EM-1 memory is highly fragmented to make adaptation to various memory architectures possible. Format and encoding of pointers is explicitly undefined. Byte addressing is concentrated in a few instructions.

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. The most straightforward layout is shown in figure 4.

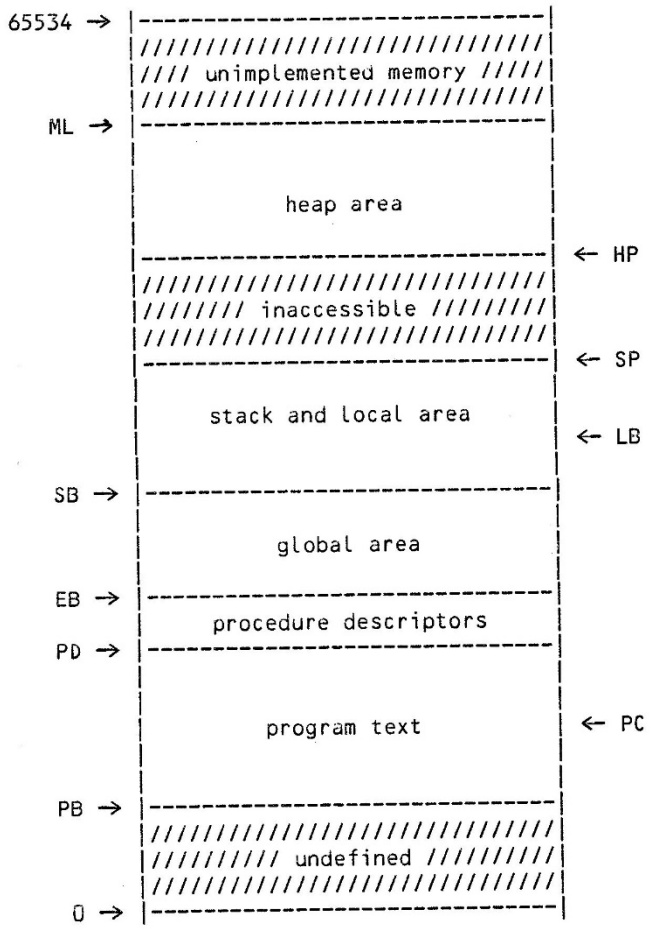


Fig. 4. Memory layout showing typical register positions during execution of an EM-1 program.

Several target machine registers are used as base registers for the various memory pieces.

PD: points to the base of the procedure descriptor table.
PB: points to the base of the instruction address space.
EB: points to the base of the data address space.
SB: points to the base of the local data area.
ML: marks the high end of the addressable data space.

The stack grows from low EM-1 addresses to high EM-1 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 the global data area.

Since EM-1 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, since EM-1 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-1 pointers are represented in the target machine as true EM-1 addresses, e.g. a pointer to EM-1 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 to it before it is used to reference 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-1 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 the value of EB, to translate all instructions that manipulate EM-1 addresses. However, the problem is not completely solved, since a compiler 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.

EM-1 requires the stack to grow from low to high EM-1 addresses. Some machines, however, have hardware PUSH and POP instructions that require the stack to grow downwards. If reasons of efficiency urge you to use these instructions, then the implementation of EM-1 may be done by implementing the memory layout shown in figure 4 upside down. This is possible because the pointer format is explicitly undefined. The first element of a word array will have a higher physical address than the second element. Problems arise for byte addressing, since even numbered bytes are still the low order bytes in a word and since words are still addressed by these low order bytes. The problem is demonstrated clearly in figure 5, type B: the bytes in a byte array allocated at EM-1 address 40 have non-linear physical addresses.

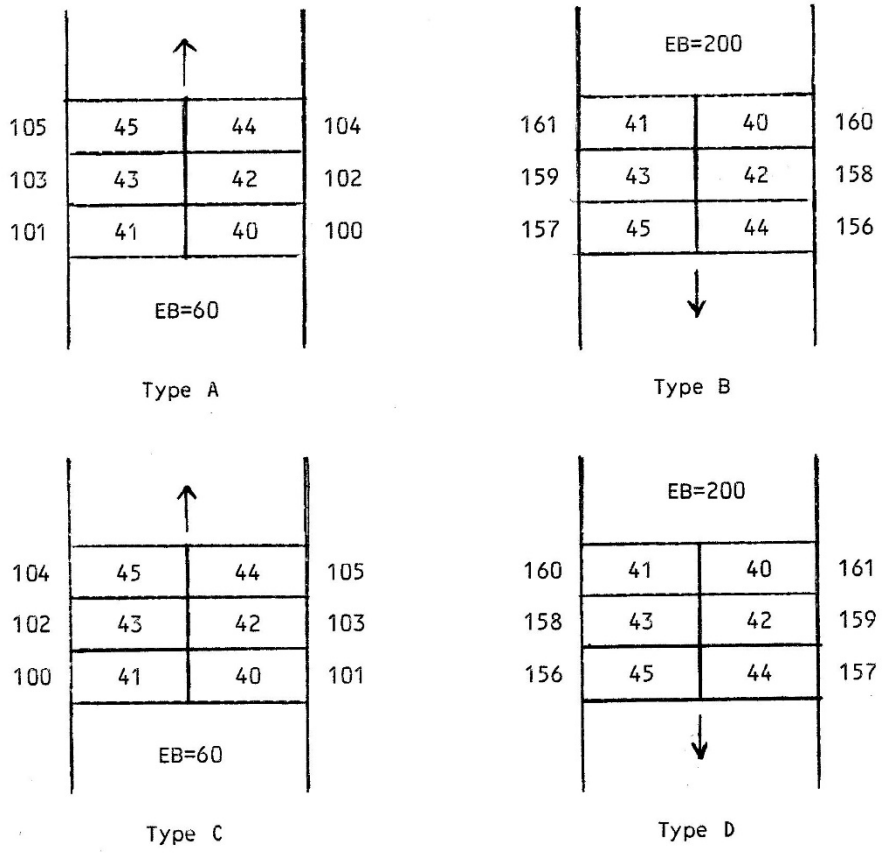


Figure 5. Four possible memory implementations.
Numbers within the boxes are EM-1 addresses.
The other numbers are physical addresses.

A similar problem may arise if EM-1 is implemented on machines for which the lowest addressed byte in a word is the high order byte, i.e. byte 0 is to the left of byte 1, instead of to the right. There are no problems with word addressing, because word addresses are still even. But byte addressing is difficult again. A fourth possible combination is to implement the EM-1 address space upside down on a machine with the bytes in a word swapped.

So, these two characteristics of the target machine - stack grows upwards or downwards, and the lowest addressed byte is the least significant byte (lsb) or the most significant byte (msb) - lead to four different EM-1 memory implementations:

A - stack upwards and lsb has lowest address
B - stack downwards and lsb has lowest address
C - stack upwards and msb has lowest address

D - stack downwards and msb has lowest address

For each of these four possibilities we give the translation of the EM-1 instructions to push the third byte of a global data block starting at EM-1 address 40 onto the stack. 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-1 instructions depends on the pointer representation used. For each of the two solutions explained above the translation is given. The second solution, a true target machine address as pointer representation, needs to be modified slightly to obtain continuous addresses for contiguous bytes. Pointers to words are true target machine addresses, but pointers to bytes may be off by two.

First the translation for the four implementations using EM-1 addresses as pointer representation:

EM-1	type A	type B	type C	type D
LAE 40	push $40	push $40	push $40	push $40
ADI 3	pop r0	pop r0	pop r0	pop r0
	add $3,r0	add $3,r0	add $3,r0	add $3,r0
	push r0	push r0	push r0	push r0
LOI 1	pop r0	pop r0	pop r0	pop r0
	-	neg r0	-	neg r0
	-	inc r0	-	inc r0
	-	xor $1,r0	xor $1,r0	-
	clr r1	clr r1	clr r1	clr r1
	bisb eb(r0),r1	bisb eb(r0),r1	bisb eb(r0),r1	bisb eb(r0),r1
	push r1	push r1	push r1	push r1

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

EM-1	type A	type B	type C	type D
LAE 40	push $eb+40	push $eb-40	push $eb+40	push $eb-40
ADI 3	pop r0 add $3,r0 push r0	pop r0 sub $3,r0 push r0	pop r0 add $3,r0 push r0	pop r0 sub $3,r0 push r0
LOI 1	pop r0 - - clr r1 bisb (r0),r1 push r1	pop r0 inc r0 xor $1,r0 clr r1 bisb (r0),r1 push r1	pop r0 - xor $1,r0 clr r1 bisb (r0),r1 push r1	pop r0 inc r0 - clr r1 bisb (r0),r1 push r1

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-1 design. The number of EM-1 instructions that is influenced by the peculiar byte addressing is limited. It only affects:

LOI/LOS, STI/STS, LAR/LAS, SAR/SAS, LAI and SAI

If EM-1 is implemented on machines with address spaces larger than 64k bytes, then there are several possibilities to use this. The simplest one is to allocate instruction and data space each in a separate 64k piece of memory. EM-1 pointers must be represented in the target machine as true EM-1 addresses, because they must fit in a single machine word. The base registers PB and EB may be loaded in hardware registers wider than 16 bits, if available.

The next possibility is to have a 32-bit instruction address space and a 16-bit data address space. The only change to the basic machine needed to achieve this is to use two word addresses within the procedure descriptors and the mark blocks and perhaps for PC. Since the size of a procedure descriptor and the size of a mark block are explicitly undefined, this change is trivial. Changing from a 16-bit to a 32-bit instruction address space does not affect existing compilers. Assemblers and back ends for the various target machines will need only minor modifications in the following instructions:

MRK/MRS, CAL/CAS, RET/RES, CSA, CSB, TRP and RTT

Therefore, the machine with enlarged instruction space is still classified as EM-1, although one of the basic design decisions (A-1) is modified. It should be pointed out, however, that 64k of program space is sufficient for rather large programs.

A third possible implementation on machines with large address spaces is described in the next chapter.

6. POSSIBLE SUCCESSORS TO EM-1

6.1. EM-2

To use the large address space available on some machines effectively, the data address space of EM-1 should be increased. The main difficulty with increasing the data space is the need for larger (multiword) addresses in many contexts. There is no technical difficulty in changing the addresses from one to two words, but the effect is to increase program size and decrease program speed considerably.

We refer to the design with 32-bit pointers for both instruction and data space as EM-2. Actually, the choice of 16 or 32 bits can be made independently for instruction space and data space, but since the price of a 32 bit instruction space is rarely more than a few hundred words in the object program, and the price of a larger data space is considerable, a large data space and a small instruction space makes little sense. The list of decisions for the EM-2 design is slightly different:

DECISION A-2: The instruction address space consists of up to 2 ** 32 bytes.
DECISION B-2: The data address space consists of up to 2 ** 32 bytes.
DECISION C-2: A machine word consists of two bytes.
DECISION D-2: The code for a single procedure must fit in 2 ** 15 bytes.
DECISION E-2: The maximum size of global data is 2 ** 15 bytes
DECISION F-2: The maximum size of a procedure frame including the operand stack is 2 ** 15 bytes.

Compilers must be changed to reserve 2 words for pointer variables (the Pascal-VU compiler has such an option already built in). Assemblers, interpreters and back ends have to be changed as well, especially for the instructions that manipulate data addresses. But, using 32-bit pointers, several alternatives are available. The simplest one is to assign a 32k memory piece to the local and global data area each. Because of the unrestricted size of the heap, the rest of the address space can be allocated to heap objects.

A second alternative is to allow each fragment in the local area to occupy up to 32k bytes. The consequences for procedure calls of this approach are described above. The local and heap area compete for memory as before, the local area growing upwards and the heap growing in the opposite direction.

Allocating 32k to each fragment in global data is not straightforward, because each word in global data must be addressable by the 15-bit offsets to the instructions like LAE. If the 32-bit address space is divided into 65536 segments, each of 64k bytes, then one can use one of these segments for the normal global data and allocate large arrays in other segments. Each word in the address space can be addressed by loading the segment base address, using the new instruction LSA, and by adding an offset to this pointer using instructions like ADI.

6.2. EM-3

Both EM-1 and EM-2 use machine words of 16 bits. The same instruction set, however, can be used for a machine, EM-3, with 32-bit machine words. Addresses into the large address spaces fit into a single word again. The design decisions for EM-3 are:

DECISION A-3: The instruction address space consists of up to 2 ** 32 bytes.
DECISION B-3: The data address space consists of up to 2 ** 32 bytes.
DECISION C-3: A machine word consists of four bytes.
DECISION D-3: The code for a single procedure must fit in 2 ** 31 bytes.
DECISION E-3: The maximum size of global data is 2 ** 31 bytes
DECISION F-3: The maximum size of a procedure frame including the operand stack is 2 ** 31 bytes.

Although the instruction set is the same as for EM-1, all compilers, optimizers, back end compilers, assemblers and interpreters have to be changed, as well as the compact assembly format and the binary machine encoding.

7. DESCRIPTORS

Besides the procedure descriptors used by the call instruction, several other instructions use descriptors, notably the range check instruction, the array instructions and the case jump instructions. Procedure descriptors are allocated in instruction space, the others in data space. The descriptors in data space may be constructed at run time, but more often they are fixed and allocated in ROM data.

Range check descriptors consist of two words:

1. lower bound
2. upper bound

The range check instruction checks a word 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.

Array descriptors describe a single dimension. For multi-dimensional arrays several array instructions in row are needed to access a single element. Array descriptors contain the following three words:

1. lower bound
2. upper bound - lower bound
3. number of bytes per element

The array instructions LAR, SAR and AAR have the offset of the start of the descriptor from the base of the global data area as operand. The array instructions LAS, SAS and AAS fetch the pointer to the descriptor from the top of the stack.

The element A[I] is fetched as follows:

a) Stack the address of A (e.g. using LAE, LEX, or LAL)
b) Stack the value of I (16 bit integer)
c) LAR n (n is offset for descriptor)

All array instructions pop the index and subtract the lower bound from it. If the result is negative, a trap occurs. If zero or positive, it is compared to upper bound - lower bound (the second descriptor word). If it is out of range, 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, which replaces A on the stack.

At this point LAR (LAS), SAR (SAS) and AAR (AAS) diverge. AAR is finished. LAR pops the address and fetches the data item the size being specified by the descriptor (this must be a multiple of the word size, except for one byte); SAR pops the address and stores the data item now exposed.

The case jump instructions CSA and CSB both provide multiway branches within a single procedure (most likely the current one), selected by a case index. Both fetch two arguments from the stack: first a pointer to 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 6, are

different.

CSA selects the new PC by indexing. The lower bound is subtracted from the case index. If the result is greater or equal to 0 and less or equal to upper-lower, then fetch the offset from the list of offsets by indexing with index-lower. If index-lower is out of bounds or if the fetched offset is -1, then fetch the default offset. If the resulting offset is -1, then trap. If not, then calculate the new PC by adding the offset to the base of the procedure text mentioned in the procedure descriptor corresponding with the procedure number. By adding the offset to the procedure base you may not come in a different procedure.

CSB selects the new PC by searching. The table is searched for an entry with index value equal to the case index. The code offset of that entry (the default offset if no such entry found) is used to calculate the new PC as for CSA.

The choice of which case instruction to use for each source language case statement is up to the compiler. 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.

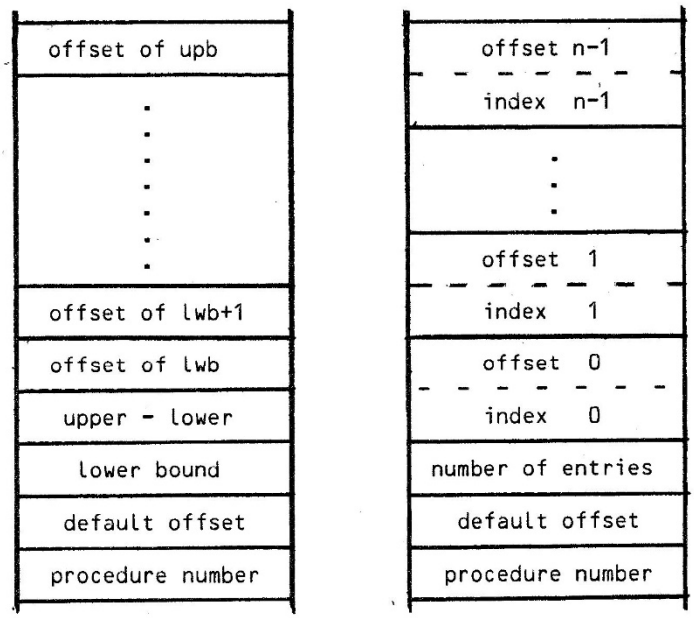


Figure 6. Descriptor layout for CSA and CSB

8. INPUT/OUTPUT AND OTHER MONITOR CALLS

EM-1 differs from most conventional machines in that it has high level i/o instructions. Typical instructions are OPEN FILE and READ FROM FILE rather than 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-1 on various non EM-1 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 p1,p2,...,pn which have been previously stacked in the order listed. Some i/o functions also provide results, which are returned on the stack. The table below lists the i/o codes and their results. This list is similar to the system calls of the UNIX operating system. (UNIX is a Trademark of Bell Laboratories.)

To execute a monitor call, proceed as follows:

a) Stack the parameters in the order specified, one word per parameter
b) Push the monitor call number (iocode) onto the stack
c) Execute the MON instruction

The result of the MON instruction can be at most 3 words: e, R1, R0. Next to the number of each call in the list below, a three character code is given. If the call returns an error code (e), the first character of the code is e, else -. The next two characters tell whether return values R1 and R0 are present (digit) or absent (-). The order in which these return values are pushed onto the stack is: R0, R1, e. So the error code, if present, is found on top of the stack. If an error occurs on a monitor call having R0, R0 is also set to the error code.

List of monitor calls.

```
 2  e10  Fork() - spawn new process
         R0 = processid of other process, R1=1 for child, 0 for parent
 3  e-0  Read(bufaddr,nbytes,fildes) - read from file
         R0 = number of bytes actually read
 4  e-0  Write(bufaddr,nbytes,fildes) - write on a file
         R0 = number of bytes actually written (should be equal to nbytes)
 5  e-0  Open(string,open flag) - open for reading or writing or both
         R0 = fildes
 6  e--  Close(fildes) - close a file
 7  e10  Wait() - wait for process to terminate
         R0 = process id , R1 = exit status
 8  e-0  Creat(string,mode) - create a new file
         R0 = fildes
 9  e--  Link(string1,string2) - link to a file
10  e--  Unlink(string) - remove directory entry
12  e--  Chdir(string) - change default directory
13  -10  Time() - get date and time
         R0 = high order word, R1 = low order word
14  e--  Mknod(string,mode,addr) - make a directory or a special file
15  e--  Chmod(string,mode) - change mode of file
16  e--  Chown(string,owner,group) - change owner and group of a file
18  e--  Stat(string,statbuf) - get file status
19  e10  Lseek(highword,lowword,fildes) - move read/write pointer
```

```
          R0 = prev. highword, R1 = prev. lowword
 20  --0  Getpid() - get process identification
          R0 = process id
 21  e--  Mount(string1,string2,rwflag) - mount file system
 22  e--  Umount(string) - unmount file system
 23  e--  Setuid(userid) - set user ID
 24  -10  Getuid() - get user ID
          R0 = real user ID , R1 = effective user ID
 25  e--  Stime(hightime,lowtime) - set time and date
 26  e-0  Ptrace(pid,addr,request,data) - process trace
          R0 = returned value
 27  --0  Alarm(seconds) - schedule signal after specified time
          R0 = previous value
 28  e--  Fstat(statbuf,fildes) - get file status
 29  ---  Pause() - stop until signal
 30  e--  Utime(string,timep) - set file times
 33  e--  Access(string,mode) - determine accessibility of file
 34  ---  Nice(incr) - set program priority
 35  ---  Ftime(bufp) - get date and time
 36  ---  Sync() - update superblock
 37  e--  Kill(sig,pid) - send signal to a process
 41  e-0  Dup(fildes,newfildes) - duplicate an open file descriptor
          R0 = new file descriptor
 42  e10  Pipe() - create an interprocess channel
          R0 = read descriptor , R1 = write descriptor
 43  ---  Times(buffer) - get process times
 44  ---  Profil(buff,bufsiz,offset,scale)- execution time profile
 46  e--  Setgid(gid) - set group ID
 47  -10  Getgid() - get group ID
          R0 = real group ID , R1 = effective group ID
 48  e-0  Sigtrp(trapno,signo) - see below
          R0 = previous value
 51  e--  Acct(file) - turn accounting on or off
 52  e--  Phys(segreg,size,physadr) - allow a process to access phys. addr.
 53  e--  Lock(flag) - lock a process in primary memory
 54  e--  Ioctl(fildes,request,argp) - control device
 56  e--  Mpxcall(cmd,vec) - manipulate multiplexed files
 59  e--  Exece(name,argv,envp) - execute a file
 60  e--  Umask(complmode) - set file creation mode mask
 61  e--  Chroot(string) - change root directory
```

Codes 0, 1, 11, 17, 31, 32, 38, 39, 40, 45, 49, 50, 55, 57, 58, 62, and 63 are not used.

The sigtrp() entry works specially. Normally, trapno is in the range 0 to 255. 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.

All of the above monitor calls, except for the sigtrp(), which is slightly modified, are the same as the UNIX version 7 system calls.

9. TRAPS AND INTERRUPTS

EM-1 provides a means for the user program to catch all errors generated by the program itself, the hardware, or external conditions. This mechanism uses three instructions: SIG, TRP and RTT. This section of the manual may be omitted on the first reading since it presupposes knowledge of the EM-1 instruction set.

The SIG instruction expects a procedure identifier on the stack, that is a static link and a procedure number. From that moment on all errors will be passed to this procedure. When an error occurs, this procedure is called with the number of the error as the only parameter (see below). SIG returns an analogous procedure identifier on the stack representing the previous procedure for dealing with errors. Two consecutive SIGs are a no-op. After the procedure is called, the values are reset to their default condition, to prevent recursive traps from hanging the machine up, e.g. stack overflow in the stack overflow handling routine. Default error processing is explicitly undefined, but will usually be the lowest possible form of getting the error number to the outside world.

The TRP instruction generates an trap, the trap number being found on the stack. This is among other things useful for library routines, runtime systems etc.. Also it can be used by a low level error routine to pass the error to a higher level one (see example below).

The RTT instruction returns from the error routine and continues after the error. In the list below all errors marked with an asterisk ('*') are considered to be fatal and it is explicitly undefined what happens if you try to restart after the error.

The way an error routine is called is completely compatible with normal calling conventions. The only way an error routine differs from normal procedures is the return. It has to use RTT instead of RET. This is necessary because all interpreter 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 255. There are three categories of error numbers:

0- 63	EM-1 machine errors, e.g. illegal instruction.
64-127	Reserved for use by compilers, run time systems, etc.
128-255	Available for user programs.

EM-1 machine errors are numbered as follows:

0*	Stack overflow
1*	Heap overflow
2*	Illegal instruction
3*	Illegal odd or zero argument
4*	Case error
5	Set bound error
6	Array bound error
7	Range bound error
8	Integer overflow
9	Double integer overflow
10	Floating overflow
11	Floating underflow

```
12   Divide by 0
13   Divide by 0.0
14   Integer undefined
15   Double integer undefined
16   Floating undefined
17   Float -> int conversion error
18   Float -> double int conversion error
19   Double int -> int conversion error
20   Floating hardware error
21   Argument of LIN too high
22   Bad monitor call
23*  CAL with wrong number of args
24   Bad argument of LAE
25*  Addressing non existent memory
26*  Bad pointer used
27*  Procedure number too high
28*  Program counter out of range
```

As an example, suppose a subroutine has to be written to generate random numbers. The usual method of doing this uses multiplications that can overflow without harm to the result of the procedure. This overflow should not be noticed by the user program since it has no idea whatsoever what happens inside the random procedure. This can be programmed as follows using the mechanism described above:

```
 let ovf_err,8          ; This is the error number of integer overflow
save bss 4              ; Room to save previous value of error routine

 pro catch,2,0          ; Local procedure that must catch the overflow trap
 lol 0                  ; Load error number
 loc ovf_err            ; check for overflow
 bne 1                  ; if other error call higher routine
 lex 1                  ; load static link to random
 loc $catch             ; load procedure number
 sig                    ; get ready to catch next overflow
 rtt                    ; return to random
1                       ; other error has occurred
 lde save               ; previous error routine
 sig                    ; other routine will get the traps now
 beg -4                 ; remove the result of sig
 lol 0                  ; stack error number
 trp                    ; call other error routine
 rtt                    ; if other routine returns, do the same
 end

 pro random,0,1         ; entry point without parameters
 lex 0                  ; static link of catch will point to LB of random so all
                        ; local variables of random are accessible from catch
 loc $catch
 sig                    ; errors will be trapped to catch
 sde save               ; save previous value
;
; calculate random number now, generating overflow at will
;
 lde save               ; restore previous status
```

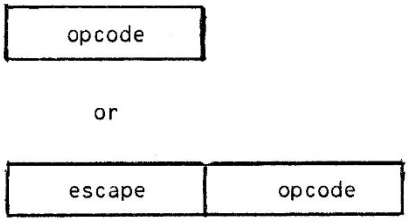
```
 sig                    ; done now
;
; load result of random
;
 ret 2                  ; return random number
 end
```

10. EM-1 MACHINE LANGUAGE

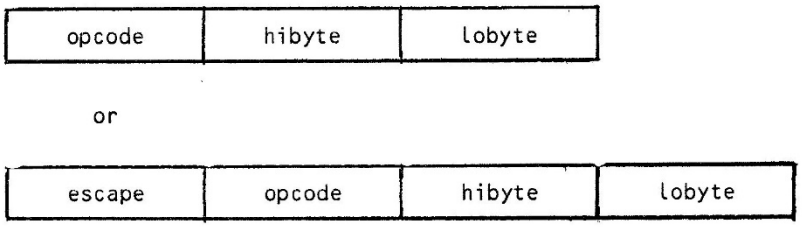
The EM-1 machine language is designed to make program text compact and to make decoding easy. Compact program text has many advantages: programs execute faster, programs occupy less primary and secondary storage and loading programs into satellite processors is faster. Since the decoding of EM-1 machine language is very simple, it is feasible to use interpreters as long as EM-1 hardware machines are not available. This chapter is irrelevant if back end compilers are used to produce executable target machine code.

10.1. INSTRUCTION ENCODING

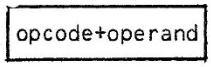
A design goal of EM-1 is to make the program text as compact as possible. Decoding must be easy, however. The encoding is fully byte oriented, without any small bit fields. There are 256 primary opcodes, one of which is an escape to a group of 256 secondary opcodes. EM-1 instructions without operands have a single opcode assigned, possibly escaped:



The situation for instructions with an operand is more complex. It is assumed that the machine words are two bytes long. There is always one opcode that takes the next two bytes as operand, high byte first (operands always fit in 16 bits):

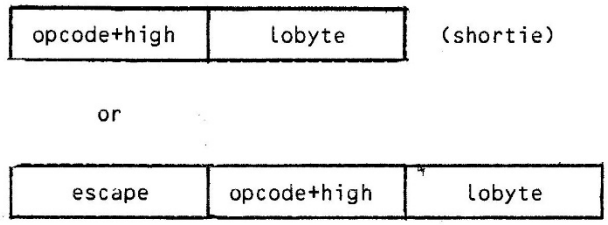


For most instructions, however, some operand values predominate. The most frequent combinations of instruction and operand will be encoded in a single byte, called a mini:



(mini)

The savings for these mini opcodes are considerable, 66% to 75%. But the number of minis is restricted, since only 255 primary opcodes are available. Many instructions have the bulk of their operands fall in the range 0 to 255. Instructions that address global data have their operands distributed over a wider range, but small values of the high byte are very common. For all these cases there is another encoding that combines the instruction and the high byte of the operand into a single opcode. These opcodes are called shorties. Shorties may be escaped.



Escaped shorties are useless if the normal encoding has a primary opcode. Note that for some instruction-operand combinations three different encodings are available. It is the task of the assembler to select the shortest of these.

Further improvements are possible: many instructions have only even operands. If these operands are divided by two, then more of them will be encoded as shortie or mini. The operands of some other instructions, that give an object size in bytes, are even, except for the value 1. In this case the value 1 is encoded as 0, since 0 is not a valid size, and the other operand values are divided by two as above.

To reduce instruction length of most call instructions to one byte, a special mechanism is available. If all the instructions between a MRK and its corresponding CAL belong to a certain group (specified below), the assembler automatically replaces the MRK by the special instruction MRX. This redefines all the opcodes fetched until the CAL has executed. The redefined opcode table is called the "alternate context". Opcodes 0 up to (but excluding) CUTOFF have the same meaning as in the regular context. However, CUTOFF means call procedure 0, CUTOFF+1 means call procedure 1, etc. Instruction MRX may only be generated by the assembler, not by compilers.

It should now be clear that the instructions allowed between MRK and CAL in alternate context are just those with opcodes 0 to CUTOFF-1. When assigning opcodes to mnemonics, those useful in parameter passing have been given low values.

Assigning opcodes to instructions by the assembler is completely table driven. For details see appendix 2.

10.2. LOAD FORMAT

The EM-1 machine language load format defines the interface between the EM-1 assembler/loader and the EM-1 machine itself. A load file consists of a header (16 words), the program text to be executed, a description of the global data area and the procedure descriptor table, in this order.

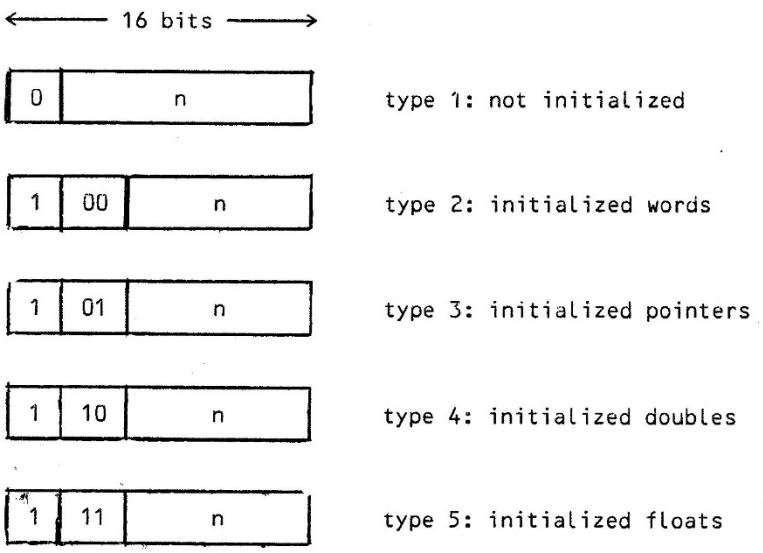
The header has two parts: the first half of it (8 words) aids in selecting the correct EM-1 machine or interpreter. Some EM-1 machines, for instance, may have hardware floating point instructions. The header words are used for:

0: magic number (07254)
1: flag bits with the following meaning:
   bit0: TEST; perform tests like integer overflow etc.
   bit1: PROFILE; for each source line: count the number of memory cycles executed.
   bit2: FLOW; for each source line: set a bit in a bitmap table if instructions on that line are executed.
   bit3: COUNT; for each source line: increment a counter if that line is entered.
   bit4: REALS; set if a program uses floating point instructions.
   bit5: EXTRA; more tests during compiler debugging.
2: number of unresolved references.
3: version number; used to detect obsolete EM-1 load files.
4: unused
5: unused
6: unused
7: unused

The second part of the header (8 words) describes the load file itself:
8: NTEXT; the program text size in bytes.
9: NDATA; the number of load file descriptors (see below).
10: NPROC; the number of entries in the procedure descriptor table.
11: ENTRY; procedure number of the procedure to start with.
12: NLINE; the maximum source line number.
13: unused
14: unused
15: unused

The program text consists of NTEXT bytes. NTEXT is always even. The first byte of the program text is the first byte of the instruction address space, i.e. it has address 0. The only pointers into the program text are found in the procedure descriptor table, so relocation is simple.

The global data area is described by the NDATA descriptors. Each descriptor describes a number of consecutive words. While reading the descriptors from the load file, one can initialize the global data area from low to high addresses. Five descriptor types are available, characterized by their first word:



type 1: Reserve n words, not explicitly initialized (BSS and HOL).
type 2: The next n words are initializers for the next n words of the global data area.
type 3: The next n words of global data are initialized pointers into the EM-1 data address space. Interpreters that represent EM-1 pointers by true EM-1 addresses do not need this information, but interpreters that represent EM-1 pointers by target machine addresses must relocate all data pointers.
type 4: The next 2*n words of global data are initialized double precision integers. The load file contains n pairs of words as initializer. The first word of each pair is the most significant one.
type 5: The load file contains n ASCII strings, null terminated, to initialize n floating point numbers in global data. Each string starts at a word boundary.

Note that the first descriptor describes the ABS block. This descriptor is of type 1.

The NPROC procedure descriptors on the load file consist of two words each: the first word gives the number of bytes for parameters, the second word gives the address of the first instruction of the procedure, assuming that program text starts at address 0.

11. EM-1 ASSEMBLY LANGUAGE

11.1. Introduction

An assembly language program consists of a series of lines, each containing 0 or 1 statements. A machine instruction may not be labeled. In other words, the label field on a machine instruction must be left blank. There are two kinds of labels, instruction and data labels. Labels start in column 1. Instruction labels are unsigned positive integers, and each must appear alone on a line by itself. The scope of an instruction label is its procedure.

The pseudoinstructions CON, ROM, and BSS may be labeled with a 1-8 character data label, the first character of which is a letter, period or underscore, followed by letters, digits, periods and underscores. Only 1 label per line is allowed. The use of the character "." followed by a number (e.g. .40) is recommended for compiler generated programs, since these are considered as a special case and handled more efficiently in compact assembly language (see below).

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.

All constants are decimal unless started with a zero e.g. 0177, in which case they are octal. In CON and ROM pseudoinstructions, floating point numbers are distinguished by the presence of a decimal point or an exponent (indicated by E or e), or both. Double precision (long) integers are followed directly by an L or l.

Also allowed as initializers in CON and ROM are strings. Strings are surrounded by double quotes and may include \xxx, where xxx is a 3-digit octal constant, e.g. CON "hello\012\000". Each string element initializes a single byte. Strings are padded at the end up to a multiple of the word size.

Local labels are referred to as *1, *2, etc. in CON and ROM pseudoinstructions (to distinguish them from constants), but without the asterisk in branch instructions, e.g. BRF 3, not BRF *3.

The notation $procname is used to mean the descriptor number for the procedure with the specified name.

An input file may contain many procedures. A procedure consists of zero or more pseudoinstructions, a PRO statement, a (possibly empty) collection of instructions and pseudoinstructions and finally an END statement. The very last statement on the input file must be EOF. The END directly preceding the EOF may be omitted.

Input to the assembler is in lower case, if available. Upper case is used in this document merely to distinguish key words from the surrounding prose.

11.2. Pseudo instructions

First the notation used for the operands of the pseudo instructions.

<num> = an integer constant
<sym> = an identifier
<arg> = <num> or <sym>
<val> = <arg>, long constant (ending with L or l), real constant, string constant (surrounded by double quotes), procedure number (starting with $) or instruction label (starting with *).
<...>* = zero or more of <...>
<...>+ = one or more of <...>

Four pseudo instructions request global data:

BSS <num>
Reserve <num> bytes, not explicitly initialized. <num> must be a multiple of the word size.

HOL <num>
Idem, but all following absolute global data references will refer to this block.

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

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

Three pseudo instructions partition the input into procedures:

PRO <sym>,<num1>,<num2>
Start of procedure. <sym> is the procedure name. <num1> is the number of bytes for arguments. <num2> is 1 for procedure names to be exported out of the current module, 0 otherwise.

END
End of Procedure.

EOF
End of module.

Besides the export flag in PRO, six other pseudo instructions are involved with separate compilation and linking:

EXD <sym>
Export data. <sym> is exported out of this module.

IMA <sym>
Import address. IMA allows global symbol <sym> to be used before it is

defined. Note that <sym> may be defined in the same module.

IMC <sym>
Similar to IMA, but used for imported single word constants. These two different forms are necessary, because the assembler must know how much storage must be allocated if <sym> is used in CON or ROM.

FWA <sym>
Forward address. Notify the assembler that <sym> will be defined later on in this module, so that it may be used before being defined.

FWC <sym>
Similar to FWA, but for constants.

FWP <sym>
Forward procedure reference. FWP allows <sym> to be used before it is defined. <sym> must be defined in the same module and must not be exported. Normally, unknown procedure names are entered in the undefined global reference table, so that their names will be known outside this module. Procedure names introduced by FWP are treated differently, however, to prevent their being exported.

Three other pseudo instructions provide miscellaneous features:

LET <sym>,<arg>
Assembly time assignment of the second operand to the first one.

EXC <num1>,<num2>
Two blocks of instructions preceding this one are interchanged before being assembled. <num1> gives the number of lines of the first block. <num2> gives the number of lines of the second one. Blank and pure comment lines do not count.

MES <num>,<val>*
A special type of comment. Used by compilers to communicate with the optimizer, assembler, etc. as follows:

MES 0 -
An error has occurred, stop assembly.

MES 1 -
Suppress optimization

MES 2 -
Use virtual memory (EM-2)

MES 3,<num1>,<num2> -
Indicates that a local variable is never referenced indirectly. <num1> is offset in bytes from LB. <num2> indicates the class of the variable.

MES 4 -
Number of source lines (for profiler).

MES 5 -
Floating point used.

MES 6,<val>* -
Comment. Used to provide comments in compact assembly language (see below).

Each back end is free to skip irrelevant MES pseudos.

11.3. 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 compiler and assembler, and also reduce the amount of storage required for storing libraries. Libraries are stored as archived compact assembly language, not machine language. The compact assembly language assumes that a machine word is two bytes long.

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
1-139	Machine instructions, see Appendix 2, alphabetical list
140-149	Reserved for future use
150-165	BSS,CON,END,EOF,EXC,EXD,FWA,FWC,FWP,HOL,IMA,IMC,LET,MES,PRO,ROM
166-179	Reserved for future pseudoinstructions
180-239	Local 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 very next byte. There are no linefeeds used to separate lines.

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

0-239	Integer constant from 0 to 239
240-255	See the Common Table below

Common Table for Neutral State and Operands

240 b1	Local label b1 (Not used for branches)
241 b1 b2	16 bit local label (256*b2 + b1)
242 b1	Global label .0-.255, with b1 being the label
243 b1 b2	Global label .0-.65535 with 256*b2+b1 being the label
244 <ASCII string>	Global symbol not of the form .nnn
245 <ASCII string>	Procedure name (not including $)
246 <ASCII string>	String used in CON or ROM (no quotes)
247 <ASCII string>	Real value for CON
248 constant 0-255	(In fact only used for 241-255 in practice)
249 b1	<negative of constant b1 in range -1 to -255>
250 b1 b2	(16 bit constant) 256*b2+b1

```
251 <ASCII string>    Double precision integer constant
255                   Delimiter for CON, ROM lists
```

The notation <ASCII string> consists first of a length field, and then an arbitrary string of bytes. If the length field starts out with 0-254, that is the length of the string. If it is 255, the length follows in the next two bytes, low order byte first.

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

```
Group 1 -- END, EOF have no operands
Group 2 -- BSS, EXC, HOL have a known number of numeric operands
Group 3 -- EXD, FWA, FWC, FWP, IMA, IMC, LET, PRO start with a string
Group 4 -- CON, MES, ROM  have a variable number of various things
```

Group 1 is easy; just go back into neutral state immediately. Groups 2 and 3 use the encoding described above. Group 4 also uses the encoding listed above, with a 255 byte after the last operand to indicate the end of the list.

Example ASCII (LOC = 76, BRF = 18 here):	Example compact
2	182
1	181
LOC 10	76 10
LOC -10	76 249 10
LOC 300	76 250 44 1
BRF 19	18 19
300	241 44 1
.3 CON 4,9,*2,$foo	242 3 152 4 9 240 2 245 3 102 111 111 255
LOC .35	76 242 35

12. ASSEMBLY LANGUAGE INSTRUCTION LIST

For each instruction in the list the range of operand values in the assembly language is given. These ranges are all subranges of -32768..32767 and are indicated by letters:

m: full range, i.e. -32768..32767
n: 0..32767
x: 0..32766 and even
y: 1 or (2..32766 and even)
z: -32768..32766 and even
p: 2..32766 and even
r: 0, 1 or 2

The letters should not be confused with the letters used in the EM-1 instruction table in appendix 2. Instructions that check for undefined operands and underflow or overflow are indicated by (*).

GROUP 1: LOAD

LOC m - Load constant (i.e. push it onto the stack)
LNC m - Load negative constant
LOL x - Load local word x
LOE x - Load external word x
LOP x - Load word pointed to by x-th local
LAI y - Load auto increment y bytes (address of pointer on stack)
LOF m - Load offsetted. (top of stack + m yield address)
LAL x - Load address of local
LAE x - Load address of external
LEX n - Load lexical. (address of LB n static levels back)
LOI y - Load indirect y bytes (address is popped from the stack)
LOS - Load indirect (pop byte count, address; count is 1 or even)
LDL x - Load double local (two consecutive locals are stacked)
LDE x - Load double external (two consecutive externals are stacked)
LDF m - Load double offsetted (top of stack + m yield address)

GROUP 2: STORE

STL x - Store local
STE x - Store external
STP x - Store into word pointed to by x-th local
SAI y - Store auto increment y bytes (address of pointer on stack)
STF m - Store offsetted
STI y - Store indirect y bytes (pop address, then data)
STS - Store indirect (pop byte count, then address, then data)
SDL x - Store double local
SDE x - Store double external
SDF m - Store double offsetted

GROUP 3: SINGLE PRECISION INTEGER ARITHMETIC

ADD - Addition (*)
SUB - Subtraction (*)
MUL - Multiplication (*)

```
  DIV   - Division (*)
  MOD   - Modulo i.e.remainder (*)
  NEG   - Negate (two's complement) (*)
  SHL   - Shift left (*)
  SHR   - Shift right (*)

GROUP 4: DOUBLE PRECISION ARITHMETIC   (Format not defined)

  DAD   - Double add (*)
  DSB   - Double Subtract (*)
  DMU   - Double Multiply (*)
  DDV   - Double Divide (*)
  DMD   - Double Modulo (*)

GROUP 5: FLOATING POINT ARITHMETIC   (Format not defined)

  FAD   - Floating add (*)
  FSB   - Floating subtract (*)
  FMU   - Floating multiply (*)
  FDV   - Floating divide (*)
  FIF   - Floating multiply and split integer and fraction part (*)
  FEF   - Split floating number in exponent and fraction part (*)

GROUP 6: POINTER ARITHMETIC

  ADI m - Add the constant m to pointer on top of stack
  PAD   - Pointer add; pop integer, then pointer, push sum as pointer
  PSB   - Subtract two pointers (in same fragment) and push diff as integer

GROUP 7: INCREMENT/DECREMENT/ZERO

  INC   - Increment top of stack by 1 (*)
  INL x - Increment local (*)
  INE x - Increment external (*)
  DEC   - Decrement top of stack by 1 (*)
  DEL x - Decrement local (*)
  DEE x - Decrement external (*)
  ZRL x - Zero local
  ZRE x - Zero external

GROUP 8: CONVERT

  CID   - Convert integer to double (*)
  CDI   - Convert double to integer (*)
  CIF   - Convert integer to floating (*)
  CFI   - Convert floating to integer (*)
  CDF   - Convert double to floating (*)
  CFD   - Convert floating to double (*)

GROUP 9: LOGICAL

  AND p - Boolean and on two groups of p bytes
  ANS   - Boolean and; number of bytes is first popped from stack
  IOR p - Boolean inclusive or on two groups of p bytes
  IOS   - Boolean inclusive or; nr of bytes is first popped from stack
```

```
  XOR p - Boolean exclusive or on two groups of p bytes
  XOS   - Boolean exclusive or; nr of bytes is first popped from stack
  COM p - Complement (one's complement of top p bytes)
  COS   - Complement; first pop number of bytes from stack
  ROL   - Rotate left
  ROR   - Rotate right

GROUP 10: SETS

  INN p - Bit test on p byte set (bit number on top of stack)
  INS   - Bit test; first pop set size, then bit number
  SET p - Create singleton p byte set with bit n on (n is top of stack)
  SES   - Create singleton set; first pop set size, then bit number

GROUP 11: ARRAY

  LAR x - Load array element
  LAS   - Load array element;  first pop ptr to descriptor from stack
  SAR x - Store array element
  SAS   - Store array element;  first pop ptr to descriptor from stack
  AAR x - Load address of array element
  AAS   - Load address;  first pop pointer to descriptor from stack

GROUP 12: COMPARE

  CMI   - Compare 2 integers. Push negative, zero, positive for <, = or >
  CMD   - Compare 2 double integers
  CMF   - Compare 2 reals
  CMU p - Compare 2 blocks of p bytes each
  CMS   - Compare 2 blocks of bytes;  pop byte count
  CMP   - Compare 2 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

  BRF n - Branch forward unconditionally n bytes
  BRB n - Branch backward unconditionally n bytes

  BLT n - Forward branch less (pop 2 words, branch if top > second)
  BLE n - Forward branch less or equal
  BEQ n - Forward branch equal
  BNE n - Forward branch not equal
  BGE n - Forward branch greater or equal
  BGT n - Forward branch greater

  ZLT n - Forward branch less than zero (pop 1 word, branch negative)
  ZLE n - Forward branch less or equal to zero
  ZEQ n - Forward branch equal zero
  ZNE n - Forward branch not zero
```

```
  ZGE n - Forward branch greater or equal zero
  ZGT n - Forward branch greater than zero

GROUP 14: PROCEDURE CALL

  MRK n - Mark stack (n = change in static depth of nesting - 1)
  MRS   - Mark stack; first pop the static link from the stack
  CAL n - Call procedure (with descriptor n)
  CAS   - Call indirect;  first pop procedure number from stack
  RET x - Return (function result consists of top x bytes)
  RES   - Like RET, but size of result on top of stack

GROUP 15: MISCELLANEOUS

  BEG z - Begin procedure (reserve z bytes for locals)
  BES   - Like BEG, except first pop z from stack
  BLM x - Block move x bytes; first pop destination addr, then source addr
  BLS   - Block move; like BLM, except first pop x, then addresses
  CSA   - Case jump; address of jump table at top of stack
  CSB   - Table lookup jump; address of jump table at top of stack
  DUP p - Duplicate top p bytes
  DUS   - Like DUP, except first pop p
  EXG   - Exchange top 2 words
  HLT   - Halt the machine ( Exit status on the stack )
  LIN n - Line number (external 0 := n)
  LNI   - Line number increment
  LOR r - Load register (0=LB, 1=SP, 2=HP)
  MON   - Monitor call
  NOP   - No operation
  RCK x - Range check; descriptor at (external) x; trap on error
  RCS   - Like RCK, except first pop x from stack
  RTT   - Return from trap
  SIG   - Trap errors to proc nr on top of stack (-2 resets default).  Static
          link of procedure is below procedure number. Old values returned
  STR r - Store register (0=LB, 1=SP, 2=HP)
  TRP   - Cause trap to occur (Error number on stack)
```

13. KERNEL INSTRUCTION SET

Many of the instructions presented in the previous chapter are replacements for a small sequence of basic instructions. The basic instructions form less than half of the complete instruction set. Only a few basic instructions have operands. Most of them fetch their arguments from the stack. Very few basic instructions are provided to load and store objects.

For each of the groups of instructions the basic ones are given:

```
GROUP  1: LOC, LAE, LEX, LOS
GROUP  2: STS
GROUP  3: ADD, SUB, MUL, DIV, SHL, SHR
GROUP  4: DAD, DSB, DMU, DDV
GROUP  5: FAD, FSB, FMU, FDV, FIF, FEF
GROUP  6: PAD, PSB
GROUP  7: -
GROUP  8: CID, CDI, CDF, CFD
GROUP  9: ANS, IOS, XOS, COS, ROL, ROR
GROUP 10: INS, SES
GROUP 11: AAS
GROUP 12: CMI, CMD, CMF, CMS, CMP, TGT, TLT, TEQ
GROUP 13: BRB, ZNE
GROUP 14: MRS, CAS, RES
GROUP 15: BES, BLS, CSA, CSB, DUS, EXG, HLT, LOR, MON, NOP, RCS,
          RTT, SIG, STR, TRP
```

Almost all the other instructions can be replaced in the assembly language by a short equivalent sequence of simpler instructions. By applying these replacements recursively a sequence of basic instructions can be found.

```
GROUP 1:
  LNC m  =  LOC -m
  LOL x  =  LAL x  +  LOI 2
  LOE x  =  LAE x  +  LOI 2
  LOP x  =  LOL x  +  LOI 2
  LAI y  =  DUP 2  +  DUP 2  +  LOI 2  +  ADI y  +  EXG    +  STI 2  +  LOI y
  LOF m  =  ADI m  +  LOI 2
  LAL x  =  LEX 0  +  ADI x
  LOI y  =  LOC y  +  LOS
  LDL x  =  LAL x  +  LOI 4
  LDE x  =  LAE x  +  LOI 4
  LDF m  =  ADI m  +  LOI 4

GROUP 2:
  STL x  =  LAL x  +  STI 2
  STE x  =  LAE x  +  STI 2
  STP x  =  LOL x  +  STI 2
  SAI y  =  DUP 2  +  DUP 2  +  LOI 2  +  ADI y  +  EXG    +  STI 2  +  STI y
  STF m  =  ADI m  +  STI 2
  STI y  =  LOC y  +  STS
  SDL x  =  LAL x  +  STI 4
  SDE x  =  LAE x  +  STI 4
  SDF m  =  ADI m  +  STI 4
```

```
GROUP 3:
  MOD     =  DUP 4  +  DIV     +  MUL     +  SUB
  NEG     =  LOC 0  +  EXG     +  SUB

GROUP 4:
  DMD     =  DUP 8  +  DDV     +  DMU     +  DSB

GROUP 6:
  ADI m   =  LOC m  +  PAD

GROUP 7:
  INC     =  LOC 1  +  ADD
  INL x   =  LOL x  +  INC     +  STL x
  INE x   =  LOE x  +  INC     +  STE x
  DEC     =  LOC 1  +  SUB
  DEL x   =  LOL x  +  DEC     +  STL x
  DEE x   =  LOE x  +  DEC     +  STE x
  ZRL x   =  LOC 0  +  STL x
  ZRE x   =  LOC 0  +  STE x

GROUP 8:
  CIF     =  CID    +  CDF
  CFI     =  CFD    +  CDI

GROUP 9:
  AND p   =  LOC p  +  ANS
  IOR p   =  LOC p  +  IOS
  XOR p   =  LOC p  +  XOS
  COM p   =  LOC p  +  COS

GROUP 10:
  INN p   =  LOC p  +  INS
  SET p   =  LOC p  +  SES

GROUP 11:
  LAR x   =  LAE x  +  LAS
  SAR x   =  LAE x  +  SAS
  AAR x   =  LAE x  +  AAS

GROUP 12:
  CMU p   =  LOC p  +  CMS
  TLE     =  TGT    +  TEQ
  TGE     =  TLT    +  TEQ
  TNE     =  TEQ    +  TEQ

GROUP 13:
  BRF n   =  LOC 0  +  ZEQ n
  BLT n   =  CMI    +  ZLT n
  BLE n   =  CMI    +  ZLE n
  BEQ n   =  CMI    +  ZEQ n
  BNE n   =  CMI    +  ZNE n
  BGE n   =  CMI    +  ZGE n
  BGT n   =  CMI    +  ZGT n
  ZLT n   =  TLT    +  ZNE n
  ZLE n   =  TLE    +  ZNE n
```

```
  ZEQ n  =  TEQ    +  ZNE n
  ZGE n  =  TGE    +  ZNE n
  ZGT n  =  TGT    +  ZNE n

GROUP 14:
  MRK n  =  LOC n  +  MRS
  CAL n  =  LOC n  +  CAS
  RET p  =  LOC p  +  RES

GROUP 15:
  BEG z  =  LOC z  +  BES
  BLM p  =  LOC p  +  BLS
  DUP p  =  LOC p  +  DUS
  LIN n  =  LOC n  +  STE 0
  LNI    =  INE 0
  RCK x  =  LAE x  +  RCS
```

The replacements for LIN and LNI are only equivalent if they precede the first HOL in that assembly module. The replacements for LAI and SAI are rather artificial. These instructions are most likely preceded by a LAL or LAE instruction. Then they replace the sequence:

```
  LAL x  +  LAI y  =  LOL x  +  DUP 2  +  ADI y  +  STL x  +  LOI y
  LAE x  +  LAI y  =  LOE x  +  DUP 2  +  ADI y  +  STE x  +  LOI y
  LAL x  +  SAI y  =  LOL x  +  DUP 2  +  ADI y  +  STL x  +  STI y
  LAE x  +  SAI y  =  LOE x  +  DUP 2  +  ADI y  +  STE x  +  STI y
```

The replacements for LAS and SAS would even be longer, because the size of the object to be loaded or stored must be fetched from the descriptor. If the size y is known, then LAS and SAS can be replaced by:

```
  LAS    =  AAS    +  LOI y
  SAS    =  AAS    +  STI y
```

14. APPENDIX 1. OFFICIAL EM-1 MACHINE DEFINITION

{ This is an interpreter for EM-1. It serves as the official machine definition. This interpreter must run on a machine which supports 32 bit arithmetic.

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 undefined whether the second operand of an ADD instruction has been popped or not if the first one is undefined (-32768).

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

4. The format of the mark block is implementation dependent.

5. The format of the procedure descriptors is implementation dependent.

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 15. The effect of a count greater than 15 or less than 0 is undefined.

}

```
program em1(tables,prog,output);

label 9999;

const
  t13   =  8192;          { 2**13    }
  t14   = 16384;          { 2**14    }
  t15   = 32768;          { 2**15    }
  t15m1 = 32767;          { 2**15 -1 }
  t16   = 65536;          { 2**16    }
  t16m1 = 65535;          { 2**16 -1 }
  t31m1 = 2147483647;     { 2**31 -1 }

  maxcode = 8191;         { highest byte in code address space }
  maxdata = 8191;         { highest byte in data address space }

  { mark block format }
  statd   = 6;            { how far is static link from lb }
  dynd    = 4;            { how far is dynamic link from lb }
  reta    = 2;            { how far is the return address from lb }
  mrksize = 6;            { size of mark block in bytes }

  { procedure descriptor format }
  pdargs  = 0;            { offset for the number of argument bytes }
  pdbase  = 2;            { offset for the procedure base }
  pdsize  = 4;            { size of procedure descriptor in bytes }

  dsize   = 4;            { size of double precision integers }
  rsize   = 4;            { size of reals }
  { header words }
  NTEXT   = 1;
  NDATA   = 2;
  NPROC   = 3;
  ENTRY   = 4;
  NLINE   = 5;

  escape  = 0;            { escape to secondary opcodes }
  undef   = -32768;       { the range of integers is -32767 to +32767 }

  { error codes }
  ESTACK  =  0; EHEAP   =  1; EILLINS =  2; EODDZ   =  3;
  ECASE   =  4; ESET    =  5; EARRAY  =  6; ERANGE  =  7;
  EIOVFL  =  8; EDOVFL  =  9; EFOVFL  = 10; EFUNFL  = 11;
  EIDIVZ  = 12; EFDIVZ  = 13; EIUND   = 14; EDUND   = 15;
  EFUND   = 16; ECFI    = 17; ECFD    = 18; ECDI    = 19;
  EFPP    = 20; ELIN    = 21; EMON    = 22; ECAL    = 23;
  ELAE    = 24; EMEMFLT = 25; EPTR    = 26; EPROC   = 27;
  EPC     = 28;
```

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

type
  bitval= 0..1;          { one bit }
  bitnr=  0..15;         { bits in machine words are numbered 0 to 15 }
  byte=   0..255;        { memory is an array of bytes }
  offset= 0..t15m1;      { positive integers are offsets }
  adr=    0..t16m1;      { a machine word interpreted as an address }
  word=  -t15..t15m1;    { a machine word interpreted as a signed integer }
  full=  -t16m1..t16m1; { intermediate results need this range }
  double=-t31m1..t31m1; { double precision range }
  bftype= (andf,iorf,xorf); { tells which boolean operator needed }
  iflags= (mini,short,xbit,ybit,zbit);
  ifset=  set of iflags;

  mnem = ( NON,
           AAR, AAS, ADD, ADI,XAND, ANS, BEG, BEQ, BES, BGE,
           BGT, BLE, BLM, BLS, BLT, BNE, BRB, BRF, CAL, CAS,
           CDF, CDI, CFD, CFI, CID, CIF, CMD, CMF, CMI, CMP,
           CMS, CMU, COM, COS, CSA, CSB, DAD, DDV, DEC, DEE,
           DEL,XDIV, DMD, DMU, DSB, DUP, DUS, EXG, FAD, FDV,
           FEF, FIF, FMU, FSB, HLT, INC, INE, INL, INN, INS,
           IOR, IOS, LAB, LAE, LAI, LAL, LAR, LAS, LDE, LDF,
           LDL, LEX, LIN, LNC, LNI, LOC, LOE, LOF, LOI, LOL,
           LOP, LOR, LOS, LSA,XMOD, MON, MRK, MRS, MRX, MUL,
           MXS, NEG, NOP, NUL, PAD, PSB, RCK, RCS, RES, RET,
           ROL, ROR, RTT, SAI, SAR, SAS, SDE, SDF, SDL, SES,
          XSET, SHL, SHR, SIG, STE, STF, STI, STL, STP, STR,
           STS, SUB, TEQ, TGE, TGT, TLE, TLT, TNE, TRP, XOR,
           XOS, ZEQ, ZGE, ZGT, ZLE, ZLT, ZNE, ZRE, ZRL);

  dispatch = record
               iflag: ifset;
               instr: mnem;
               implicit: word
             end;


var
  code: packed array[0..maxcode] of byte;      { code space }
  data: packed array[0..maxdata] of byte;      { data space }
  pc,lb,sp,hp,pd: adr;  { internal machine registers }
  i: integer;           { integer scratch variable }
  s,t,k: word;          { scratch variables }
  j:offset;             { 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 }
  escaped: boolean;     { true for escaped opcodes }
  cutoff: byte;         { opcode of first call in alternate context }
  dispat: array[boolean,byte] of dispatch;
```

```
  insr: mnem;           { holds the instructionnumber }
  normalmap: boolean;   { true except when in alternate context }
  halted: boolean;      { normally false. set to true by halt instruction }
  exitstatus:word;      { parameter of HLT }
  uerrorlb:adr;         { static link of error procedure }
  uerrorproc:adr;       { number of user defined error procedure }
  header: array[1..8] of adr;

  tables: text;         { description of EM-1 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 oddchkadr(a:adr);
begin if (a>maxdata) or ((a>sp) and (a<hp)) then trap(EPTR) end;

procedure chkadr(a:adr);
begin if odd(a) then trap(EPTR); oddchkadr(a) end;

procedure newpc(a:adr);
begin if (a<0) or (a>pd) then trap(EPC); pc:=a end;

procedure newsp(a:adr);
begin if (a<lb-2) or (a>=hp) or odd(a) then trap(ESTACK); sp:=a end;

procedure newlb(a:adr);
begin if (a>sp+2) or odd(a) then trap(ESTACK); lb:=a end;

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

function argi(w:word):word;
begin if w = undef then trap(EIUND); argi:=w end;

function argn(w:word):word;
begin if w<0 then trap(EILLINS); argn:=w end;

function argx(w:word):word;
begin if (w<0) or (w>=t15) or odd(w) then trap(EILLINS); argx:=w end;

function argp(w:word):word;
begin if odd(w) or (w<=0) or (w>=t15) then trap(EILLINS); argp:=w end;

function argy(w:word):word;
```

```
begin if w=1 then argy:=1 else argy:=argp(w) end;

function argz(w:word):word;
begin if odd(w) or (w<-t15) or (w>=t15) then trap(EILLINS); argz:=w end;

function chkovf(z:double):word;
begin if abs(z) >= t15 then trap(EIOVFL); chkovf:=z end;




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

{ memw returns a machine word as a signed integer: -32768 <= memw <= +32767
  mema returns a machine word as an address : 0 <= mema <= 65535
  memb returns a single byte as a positive integer: 0 <= memb <= 255
  store(a,v) stores the word or address v at machine address a
  storeb(a,b) stores the byte b at machine address a

  memi returns a word from the instruction space: 0 <= memi <= 65535
       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.

  All routines check to make sure the address is within range.  The word
  routines also check to see that the address is even.  If an addressing
  error is found, a trap occurs. }


function mema(a:adr):adr;
var b:adr;
begin chkadr(a); b:=data[a+1]; mema:=256*b + data[a] end;

function memw(a:adr):word;
var b:adr;
begin b:=mema(a); if b>=t15 then memw:=b-t16 else memw:=b end;

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

procedure store(a:adr; x:full);
begin chkadr(a);
  if x < 0 then x := x+t16; { equivalent value, but positive }
  data[a] := x mod 256; data[a+1] := x div 256
end;

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

function memi(a:adr):adr;
```

```
var b:adr;
begin
  if odd(a) or (a>maxcode) then trap(EPTR);
  b:=code[a+1]; memi:=256*b + code[a]
end;

function nextpc:byte;
begin nextpc:=code[pc]; newpc(pc+1) end;

procedure lino(w:word);
begin if (w<0) or (w>header[NLINE]) then trap(ELIN); store(0,w) end;




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

{ push puts a word or address on the stack
  popw removes a machine word from the stack and delivers it as a word
  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 2 machine words and returns a double precision integer
  pushr pushes a real (floating point) number onto the stack
  popr removes 2 machine words and returns a real number
  pushx puts an object of arbitrary size on the stack
  popx removes an object of arbitrary size
                                                                        }

procedure push(x:full);
begin newsp(sp+2); store(sp,x) end;

function popw:word;
begin popw:=memw(sp); newsp(sp-2) end;

function popa:adr;
begin popa:=mema(sp); newsp(sp-2) end;

procedure pushd(y:double);
begin { push double integer onto the stack } newsp(sp+dsize) end;

function popd:double;
begin { pop double integer from the stack } newsp(sp-dsize); popd:=0 end;

procedure pushr(z:real);
begin { Push a real onto the stack } newsp(sp+rsize) end;

function popr:real;
begin { pop real from the stack } newsp(sp-rsize); popr:=0.0 end;

procedure pushx(size:offset; a:adr);
var i:integer;
begin
```

```
  if size=1
     then push(memb(a))
     else if odd(size) or (size<=0)
             then trap(EODDZ)
             else for i:=1 to size div 2 do push(memw(a-2+2*i))
end;

procedure popx(size:offset; a:adr);
var i:integer;
begin
  if size=1
     then begin storeb(a,memb(sp)); newsp(sp-2) end
     else if odd(size) or (size<=0)
             then trap(EODDZ)
             else for i:=1 to size div 2 do store(a+size-2*i,popw)
end;




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

procedure sleft(var w:word);  { 1 bit left shift   }
begin if abs(w) >= t14 then trap(EIOVFL) else w := 2*w end;

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

procedure rleft(var w:word);  { 1 bit left rotate }
begin if w >= 0
        then if w < t14 then w:= 2*w else w:= 2*w-t16
        else if w >= -t14 then w := 2*w+1 else w:= 2*w+t16+1
end;

procedure rright(var w:word);  { 1 bit right rotate }
begin if odd(w)
        then if w<0 then w:=(w-1) div 2 else w := w div 2 - t15
        else if w<0 then w:=(w+t16) div 2 else w:= w div 2
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:=ord(odd(w)) end;

function bf(ty:bftype; w1,w2:word):word;  { return boolean fcn of 2 words }
var i:bitnr; j:adr;
begin j:=0;
      for i:= 15 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;
      if j <= t15m1 then bf:=j else bf:= j - t16
end;

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

function arraycalc(c:adr):adr; { subscript calculation }
var j:word; size:offset; a:adr;
begin j:= popw - memw(c);
  if (j<0) or (j>memw(c+2)) then trap(EARRAY);
  size := memw(c+4);
  if (size<0) or ((size>1) and odd(size)) then trap(EODDZ);
  a := j*size+popa;
  arraycalc:=a
end;




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

{ All routines for doubles and reals are dummy routines, since the format of
  doubles and reals is not defined in EM-1.
}

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

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

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

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

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

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

function dofsb(x,y:real):real;
begin { subtract two reals } dofsb:=0.0 end;

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

```
function dofdv(x,y:real):real;
begin { divide two reals } dofdv:=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 }
  fraction:=0.0; { fractional part of x*y }
end;

procedure dofef(x:real;var mantissa:real;var exponent:integer);
begin { dismember x into mantissa and exponent parts }
  mantissa:=0.0;  { mantissa of x }
  exponent:=0;    { exponent of x }
end;
```

```
{------------------------------------------------------------------------}
{                                Trap                                    }
{------------------------------------------------------------------------}

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 uerrorlb=0 then
    begin
      writeln('error ', n:1, ' occurred without being caught');
      goto 9999
    end;
  { Deposit all interpreter variables that need to be saved on
    the stack. This includes normalmap, 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 will 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.

    For simplicity just the normalmap flag will be stacked here }

  push(ord(normalmap));
  { Now simulate the effect of an MRS instruction }
  push(uerrorlb);                { push static link }
  push(lb);                      { push dynamic link }
  push(pc);                      { push return address }
  push(n);                       { push error number }
  { Now simulate the effect of a CAS instruction }
  newlb(sp); newpc(memi(pd+pdsize*uerrorproc+pdbase));
  if n in [ESTACK,EHEAP,EILLINS,EODDZ,ECASE,ECAL,EMEMFLT,EPTR,
           EPROC,EPC]
     then goto 9999;
end;

procedure dortt;
var s:adr;
begin
  newpc(mema(lb-reta)); s:=lb-mrksize-2; newlb(mema(lb-dynd)); newsp(s);
  { So far this was a plain ret 0 }
  normalmap := popw = 1;
end;
```

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

procedure initialize;  { start the ball rolling }
{ This is not part of the official machine definition }
const tab = '    ';
var b:boolean;
    cset:set of char;
    f:ifset;
    nmini,mbase,nshort,sbase,obase,i,j,n:integer;
    c:char;

    function readword:word;
    var b1,b2:byte; a:adr;
    begin read(prog,b1,b2); a:=b2; a:=b1+256*a;
      if a>=t15 then readword:=a-t16 else readword:=a
    end;

    function readdouble:double;
    var a,b:adr;
    begin a:=readword; b:=readword;
      { construct double out of a and b } readdouble:=0
    end;

    function readreal:real;
    var b:byte; i:integer;
        s:array[1..100] of char;
    begin i:=0;
      repeat
        read(prog,b); i:=i+1; s[i]:=chr(b)
      until b=0;
      if odd(i) then read(prog,b);  { skip padding byte }
      { construct real out of character string s } readreal:=0.0
    end;

begin
  normalmap:=true;
  halted:=false;
  exitstatus:=-1;
  uerrorlb:=0;
  uerrorproc:=0;

  { initialize tables }
  for i:=0 to maxcode do code[i]:=0;
  for i:=0 to maxdata do data[i]:=0;
  for b:=false to true do
    for i:=0 to 255 do
      with dispat[b][i] do
        begin instr:=NON; iflag:=[zbit] end;

  { read instruction table file. see appendix 2 }
  reset(tables); insr:=NON;
  repeat readln(tables) until eoln(tables);     { skip until empty line }
  repeat readln(tables) until eoln(tables);     { skip until empty line }
```

```
readln(tables);                                    { skip empty line }
repeat
  insr:=succ(insr); cset:=[]; f:=[];
  read(tables,c,c,c,c);
  while (c=' ') or (c=tab) do read(tables,c);
  repeat
    cset:=cset+[c];
    read(tables,c)
  until (c=' ') or (c=tab);
  readln(tables,nmini,mbase,nshort,sbase,obase);
  if 'x' in cset then f:=f+[xbit];
  if 'y' in cset then f:=f+[ybit];
  if 'z' in cset then
    with dispat['s' in cset][obase] do
      begin iflag:=f+[zbit]; instr:=insr end
  else
    begin
      with dispat['l' in cset][obase] do
        begin iflag:=f; instr:=insr end;
      for i:=0 to nshort-1 do
        with dispat['s' in cset][sbase+i] do
          begin iflag:=f+[short]; instr:=insr; implicit:=256*i end;
      if insr=CAL then cutoff:=mbase else
        for i:=0 to nmini-1 do
          with dispat[false][mbase+i] do
            begin iflag:=f+[mini]; instr:=insr;
              implicit:=i+ord('o' in cset)
            end;
    end;
until eoln(tables);

{ read in program text, data and procedure descriptors }
reset(prog);
for i:=1 to 8 do n:=readword;  { skip first header }
for i:=1 to 8 do header[i]:=readword;  { read second header }
lb:=0; hp:=maxdata+1; sp:=0; lino(0);
{ read program text }
for i:=1 to header[NTEXT] do read(prog, code[i-1]);
{ read data blocks }
for i:=2 to readword do push(undef);  { ABS block }
for i:=2 to header[NDATA] do
  begin n:=readword;
    if n>=0 then
      for j:=1 to n do push(undef)
    else
      begin j:=(n+t15) div t13; n:=(n+t15) mod t13;
        case j of
        0, { words }
        1: { pointers }
           for j:=1 to n do push(readword);
        2: { double integers }
           for j:=1 to n do pushd(readdouble);
        3: { reals as character strings }
           for j:=1 to n do pushr(readreal);
        end
```

```
        end
    end;
  { read descriptor table }
  pd:=header[NTEXT];
  for i:=1 to header[NPROC]*pdsize do read(prog,code[pd+i-1]);
  { call the entry point routine }
  push(maxdata);  { illegal static link }
  push(maxdata);  { illegal dynamic link }
  push(maxcode);  { illegal return address }
  newlb(sp+2);
  newpc(memi(pd + pdsize*header[ENTRY] + pdbase));
end;
```

```
{------------------------------------------------------------------------}
{                        MAIN LOOP OF THE INTERPRETER                    }
{------------------------------------------------------------------------}
```

{ It should be noted that the interpreter (microprogram) for an EM-1 machine can be written in two fundamentally different ways: (1) the instruction operands are fetched in the main loop, or (2) the instruction operands are fetched after the 256 way branch, by the execution 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)

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-1 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 production 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 indirectly to a register that contains the address of the main loop. When the alternate context is entered (after the MRX or MXS instructions), this register is reloaded so that an alternate main loop is used, with an alternate branch table. 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 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 instructions 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:

```
30: LOL 0
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 6 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 em1 program counter.
Register r4 is the em1 LB register
Register r3 is the em1 SP register (the stack grows toward high 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 6 sample instructions given above might be as follows:

```
lol0: mov (r4),(sp)+       /push local 0 onto stack
      jmp (r2)             /go back to main loop
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: clr 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:

lol0, 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 }

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

begin initialize;
  repeat
    opcode := nextpc;       { fetch the first byte of the instruction }
    if normalmap or (opcode<cutoff) then
      begin escaped:=opcode=escape;
        if escaped then opcode := nextpc;
        with dispat[escaped][opcode] do
          begin insr:=instr;
            if not (zbit in iflag) then
              begin
                if mini in iflag then k:=implicit else
                if short in iflag then k:=implicit+nextpc else
                  begin k:=nextpc; if k>=128 then k:=k-256;
                        k:=256*k + nextpc
                  end;
                if xbit in iflag then k:=k*2 else
                if ybit in iflag then
                  if k=0 then k:=1 else k:=k*2
              end
          end
      end
    else
      begin insr:=CAL; k:=opcode-cutoff end;



{-------------------------------------------------------------------------}
{           Routines for the individual instructions                      }
{-------------------------------------------------------------------------}

case insr of

  NON: trap(EILLINS);

  { LOAD GROUP }
  LOC: push(k);
  LNC: push(-k);
  LOL: push(memw(lb+argx(k)));
  LOE: push(memw(argx(k)));
  LOP: push(memw(mema(lb+argx(k))));
  LAI: begin k:=argy(k); a:=popa; b:=mema(a); store(a,b+k); pushx(k,b) end;
  LOF: push(memw(popa+k));
  LAL: push(lb+argx(k));
  LAE: push(argx(k));
  LEX: begin a:=lb; for j:=1 to argn(k) do a:= mema(a-statd); push(a) end;
  LOI: pushx(argy(k),popa);
  LOS: begin k:=popa; pushx(argy(k),popa) end;
  LDL: begin k:=argx(k); push(memw(lb+k)); push(memw(lb+k+2)) end;
  LDE: begin k:=argx(k); push(memw(k)); push(memw(k+2)) end;
  LDF: begin a:=popa; push(memw(a+k)); push(memw(a+k+2)) end;
```

```
{ STORE GROUP }
STL: store(lb+argx(k),popw);
STE: store(argx(k),popw);
STP: store(mema(lb+argx(k)),popw);
SAI: begin k:=argy(k); a:=popa; b:=mema(a); store(a,b+k); popx(k,b) end;
STF: begin a:=popa; store(a+k,popw) end;
STI: popx(argy(k),popa);
STS: begin k:=popa; popx(argy(k),popa) end;
SDL: begin k:=argx(k); store(lb+k+2,popw); store(lb+k,popw) end;
SDE: begin k:=argx(k); store(k+2,popw); store(k,popw) end;
SDF: begin a:=popa; store(a+2+k,popw); store(a+k,popw) end;


{ SINGLE PRECISION ARITHMETIC }
ADD: begin t:=argi(popw); s:= argi(popw); push(chkovf(s+t)) end;
SUB: begin t:=argi(popw); s:= argi(popw); push(chkovf(s-t)) end;
MUL: begin t:=argi(popw); s:= argi(popw); push(chkovf(s*t)) end;
XDIV: begin t:= argi(popw); s:= argi(popw);
           if t=0 then trap(EIDIVZ) else push(s div t)
      end;
XMOD: begin t:= argi(popw); s:=argi(popw);
           if t=0 then trap(EIDIVZ) else push(s - (s div t)*t)
      end;
 NEG: begin t:=argi(popw); push(-t) end;
 SHL: begin t:=argi(popw); s:=argi(popw);
        for i:= 1 to t do sleft(s); push(s)
      end;
 SHR: begin t:=argi(popw); s:=argi(popw);
        for i:= 1 to t do sright(s); push(s)
      end;


{ DOUBLE PRECISION ARITHMETIC }
DAD: begin dt:=popd; ds:=popd; pushd(dodad(ds,dt)) end;
DSB: begin dt:=popd; ds:=popd; pushd(dodsb(ds,dt)) end;
DMU: begin dt:=popd; ds:=popd; pushd(dodmd(ds,dt)) end;
DDV: begin dt:=popd; ds:=popd; pushd(doddv(ds,dt)) end;
DMD: begin dt:=popd; ds:=popd; pushd(dodmd(ds,dt)) end;


{ FLOATING POINT ARITHMETIC }
FAD: begin rt:=popr; rs:=popr; pushr(dofad(rs,rt)) end;
FSB: begin rt:=popr; rs:=popr; pushr(dofsb(rs,rt)) end;
FMU: begin rt:=popr; rs:=popr; pushr(dofmu(rs,rt)) end;
FDV: begin rt:=popr; rs:=popr; pushr(dofdv(rs,rt)) end;
FIF: begin rt:=popr; rs:=popr; dofif(rt,rs,x,y); pushr(y); pushr(x) end;
FEF: begin rt:=popr; dofef(rt,x,i); pushr(x); push(i) end;


{ POINTER ARITHMETIC }
ADI: push(popa+k);
PAD: begin t:=popw; push(popa+t) end;
PSB: begin a:=popa; b:=popa; push(chkovf(b-a)) end;
```

```
{ INCREMENT/DECREMENT/ZERO }
INC: push(chkovf(argi(popw)+1));
INL: begin k:=argx(k); t:=argi(memw(lb+k)); store(lb+k,chkovf(t+1)) end;
INE: begin k:=argx(k); t:=argi(memw(k)); store(k,chkovf(t+1)) end;
DEC: push(chkovf(argi(popw)-1));
DEL: begin k:=argx(k); t:=argi(memw(lb+k)); store(lb+k,chkovf(t-1)) end;
DEE: begin k:=argx(k); t:=argi(memw(k)); store(k,chkovf(t-1)) end;
ZRL: store(lb+argx(k),0);
ZRE: store(argx(k),0);


{ CONVERT GROUP }
CID: pushd(popw);
CDI: begin dt:=popd; if abs(dt) > t15m1 then trap(ECDI) else push(dt) end;
CIF: pushr(popw);
CFI: begin rt:=popr;
       if abs(rt)>t15m1-0.5 then trap(ECFI) else push(round(rt))
     end;
CDF: begin dt:=popd; pushr(dt) end;
CFD: begin rt:=popr; if abs(rt) > t31m1-0.5 then trap(ECFD) ;
       pushd( round(rt) )
     end;


{ LOGICAL GROUP }
XAND,ANS:
     begin if insr=ANS then k:=popw; k:=argp(k);
       for j:= 1 to k div 2 do
         begin t:=popw; a:=sp-k+2; store(a,bf(andf,memw(a),t)) end;
     end;
IOR,IOS:
     begin if insr=IOS then k:=popw; k:=argp(k);
       for j:= 1 to k div 2 do
         begin t:=popw; a:=sp-k+2; store(a,bf(iorf,memw(a),t)) end;
     end;
XOR,XOS:
     begin if insr=XOS then k:=popw; k:=argp(k);
       for j:= 1 to k div 2 do
         begin t:=popw; a:=sp-k+2; store(a,bf(xorf,memw(a),t)) end;
     end;
COM,COS:
     begin if insr=COS then k:=popw; k:=argp(k);
       for j:= 1 to k div 2 do
         begin store(sp-k+2*j, bf(xorf,memw(sp-k+2*j), -1)) end
   . end;
ROL: begin t:=popw; s:=popw; for i:= 1 to t do rleft(s); push(s) end;
ROR: begin t:=popw; s:=popw; for i:= 1 to t do rright(s); push(s) end;


{ SET GROUP }
INN,INS:
     begin if insr=INS then k:=popw; k:=argp(k);
       t:=popw; if t<0 then trap(ESET);
       i:= t mod 16; t:= t div 16; if 2*t>=k then trap(ESET);
       s:=memw(sp-k+2+2*t); newsp(sp-k); push(bit(i,s));
```

```
     end;
XSET,SES:
     begin if insr=SES then k:=popw; k:=argp(k);
       t:=popw; if t<0 then trap(ESET);
       i:= t mod 16; t:= t div 16; if 2*t>=k then trap(ESET);
       for j:= 1 to t do push(0);
       s:=1; for j:= 1 to i do rleft(s); push(s);
       for j := 1 to k div 2-t-1 do push(0)
     end;


{ ARRAY GROUP }
LAR,LAS:
     begin if insr=LAS then k:=popa; k:=argx(k);
       pushx(memw(k+4),arraycalc(k))
     end;
SAR,SAS:
     begin if insr=SAS then k:=popa; k:=argx(k);
       popx(memw(k+4),arraycalc(k))
     end;
AAR,AAS:
     begin if insr=AAS then k:=popa; k:=argx(k);
       push(arraycalc(k))
     end;


{ COMPARE GROUP }
CMI: begin t:=popw; s:=popw;
       if s<t then push(-1) else if s=t then push(0) else push(1)
     end;
CMP: begin a:=popa; b:=popa;
      if b<a then push(-1) else if b=a then push(0) else push(1)
     end;
CMD: begin dt:=popd; ds:=popd;
       if ds<dt then push(-1) else if ds=dt then push(0) else push(1)
     end;
CMF: begin rt:=popr; rs:=popr;
       if rs<rt then push(-1) else if rs=rt then push(0) else push(1)
     end;
CMU,CMS:
     begin if insr=CMS then k:=popw; k:=argp(k);
       t:= 0; j:= 0;
       while (j < k) and (t=0) do
         begin a:= mema(sp-j); b:=mema(sp-k-j);
           if b<a then t:= -1 else if b>a then t:= 1;
           j:=j+2
         end;
       newsp(sp-2*k); push(t);
     end;

TLT: if popw <  0 then push(1) else push(0);
TLE: if popw <= 0 then push(1) else push(0);
TEQ: if popw =  0 then push(1) else push(0);
TNE: if popw <> 0 then push(1) else push(0);
TGE: if popw >= 0 then push(1) else push(0);
```

```
TGT: if popw >  0 then push(1) else push(0);


{ BRANCH GROUP }
BRF: newpc(pc+argn(k));
BRB: newpc(pc-argn(k));

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

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


{ PROCEDURE CALL GROUP }

{ There are four ways to mark the  stack.  The change in static depth can
  be given as an immediate operand or the new static link can be provided
  on the stack.  Also, the instruction may switch into alternate context,
  or not. Only two of these have mnemonics, i.e. can be used by the prog-
  rammer. These mnemonics are MRK and MRS, corresponding to the immediate
  and stacked forms respectively. The decision about using alternate con-
  text is made by the assembler.  The four cases are:
     MRK: immediate, normal context
     MRX: immediate, alternate context
     MRS: stacked, normal context
     MXS: stacked, alternate context
}

MRK,MRS,MRX,MXS:
     begin if (insr=MRS) or (insr=MXS) then k:=popw; k:=argn(k);
       a:= lb; for j:= 1 to k do a:= mema(a-statd);
       push(a); push(lb); push(0);
       normalmap:=(insr=MRK) or (insr=MRS);
     end;
CAL,CAS:
     begin if insr=CAS then k:=popw; k:=argn(k);
       a:=pd+pdsize*k; t:= memi(a+pdargs); store(sp+2-t-reta,pc);
       newpc(memi(a+pdbase)); newlb(sp+2-t); normalmap:=true;
     end;
RET,RES:
     begin if insr=RES then k:=popw; k:=argx(k);
       newpc(mema(lb-reta)); a:=sp-k; b:=lb-mrksize-2;
       newlb(mema(lb-dynd));
       for j:= 1 to k div 2 do store(b+2*j,memw(a+2*j));
       newsp(b+k);
     end;
```

```
{ MISCELLANEOUS GROUP }
BEG,BES:
     begin if insr=BES then k:=popw; k:=argz(k);
       if k>=0
         then for j:= 1 to k div 2 do push(undef)
         else newsp(sp+k);
     end;
BLM,BLS:
     begin if insr=BLS then k:=popw; k:=argx(k);
       t:=popa; s:=popa;
       for j := 1 to k div 2 do store(t-2+2*j,memw(s-2+2*j))
     end;
CSA: begin k:=popa; b:=memi(pd+pdsize*memw(k)+pdbase);
       t:= popw - memw(k+4); s:=-1;
       if (t>=0) and (t<=memw(k+6)) then s:=memw(k+8+2*t);
       if s=-1 then s:=memw(k+2);
       if s=-1 then trap(ECASE) else newpc(b+s)
     end;
CSB: begin k:=popa; b:=memi(pd+pdsize*memw(k)+pdbase);
       t:=popw; i:=1; found:=false;
       while (i<=memw(k+4)) and not found do
         if t=memw(k+2+4*i) then found:=true else i:=i+1;
       if found then s:=memw(k+4+4*i) else s:=memw(k+2);
       if s=-1 then trap(ECASE) else newpc(b+s);
     end;
DUP,DUS:
     begin if insr=DUS then k:=popw; k:=argp(k);
       for i:=1 to k div 2 do push(memw(sp - k + 2));
     end;
EXG: begin t:=popw; s:=popw; push(t); push(s) end;
HLT: begin exitstatus:=popw; halted := true end;
LIN: lino(argn(k));
LNI: lino(memw(0)+1);
LOR: begin i:=k;
       case i of 0:push(lb); 1:push(sp); 2:push(hp) end;
     end;
MON: ; { MON will not be described here }
NOP: ;
RCK,RCS:
     begin if insr=RCS then k:=popa; k:=argx(k);
       if (memw(sp)<memw(k)) or (memw(sp)>memw(k+2)) then trap(ERANGE)
     end;
RTT: dortt;
SIG: begin a:=popa; b:=popa; push(uerrorlb); push(uerrorproc);
       uerrorproc:=a; uerrorlb:=b
     end;
STR: begin i:=k;
       case i of 0: newlb(popa); 1: newsp(popa); 2: newhp(popa) end;
     end;
TRP: trap(popw);


  end  { end of case statement }
 until halted;
9999:
```

```
  writeln('halt with exit status:',exitstatus);
end.
```

15. APPENDIX 2. EM-1 CODE TABLES

Many programs involved with EM-1 are table driven. Some of these tables are presented in this appendix. The first table contains some constants describing the compact assembly code format:

sp_fmnem	1
sp_nmnem	149
sp_lmnem	139
sp_fpseu	150
sp_npseu	30
sp_lpseu	165
sp_filb0	180
sp_nilb0	60
sp_fcst0	0
sp_ncst0	240
sp_ilb1	240
sp_ilb2	241
sp_dlb1	242
sp_dlb2	243
sp_dnam	244
sp_pnam	245
sp_scon	246
sp_rcon	247
sp_cst1	248
sp_cstm	249
sp_cst2	250
sp_lcon	251
sp_cend	255

The next table gives the numbers of the assembly language pseudo instructions:

ps_bss	0
ps_con	1
ps_end	2
ps_eof	3
ps_exc	4
ps_exd	5
ps_fwa	6
ps_fwc	7
ps_fwp	8
ps_hol	9
ps_ima	10
ps_imc	11
ps_let	12
ps_mes	13
ps_pro	14
ps_rom	15

The third table gives the list of EM-1 instructions, sorted alphabetically. The mnem field gives the 3 character mnemonic of the instruction, the flags

field gives various flags as explained below, then follow the number of opcodes encoded with the instruction in one byte, the base of these minis, the number of opcodes wherein the highbyte is encoded and the base of these opcodes, and finally the opcode for all other instructions with that mnemonic. The flag field contains letters, describing some features of the instructions. Three groups of features are described:

Instruction type (mutually exclusive):
- a - Alternate context mark instruction
- b - Branch instruction
- c - Call instruction
- e - Operand is offset into global data
- m - Normal context mark instruction

Operand type (mutually exclusive):
- x - Only even operands allowed
- y - Only 1,2,4,6,8,.... allowed as operands
- z - No operands

Encoding information:
- s - Escaped opcodes
- l - Only the long format escaped
- o - Minis start at 1, not at 0

Now the EM-1 instruction list sorted alphabetically:

NUMBER	MNEM	FLAGS	NMINIS	MBASE	NSHORT	SBASE	OBASE
1	aar	xe	0	0	2	138	140
2	aas	zs	0	0	0	0	1
3	add	z	0	0	0	0	129
4	adi	l	0	0	1	156	2
5	and	slxo	1	229	1	4	5
6	ans	zs	0	0	0	0	6
7	beg	lxo	3	252	1	255	7
8	beq	lbo	0	0	1	165	8
9	bes	zs	0	0	0	0	16
10	bge	lbo	0	0	1	166	9
11	bgt	lbo	0	0	1	167	10
12	ble	lbo	0	0	1	168	11
13	blm	lx	0	0	1	173	17
14	bls	zs	0	0	0	0	18
15	blt	lbo	0	0	1	169	12
16	bne	lbo	0	0	1	170	13
17	brb	lbo	0	0	1	171	15
18	brf	lbo	0	0	1	172	14
19	cal	lc	122	134	1	174	19
20	cas	zsc	0	0	0	0	20
21	cdi	zs	0	0	0	0	21
22	cdf	zs	0	0	0	0	100
23	cfd	zs	0	0	0	0	101
24	cfi	z	0	0	0	0	175
25	cid	zs	0	0	0	0	22

26	cif	z	0	0	0	0	157
27	cmd	zs	0	0	0	0	23
28	cmf	z	0	0	0	0	176
29	cmi	z	0	0	0	0	177
30	cmp	zs	0	0	0	0	84
31	cms	zs	0	0	0	0	24
32	cmu	slx	0	0	1	25	26
33	com	slx	0	0	1	27	28
34	cos	zs	0	0	0	0	29
35	csa	z	0	0	0	0	178
36	csb	z	0	0	0	0	179
37	dad	zs	0	0	0	0	30
38	ddv	zs	0	0	0	0	33
39	dec	z	0	0	0	0	134
40	dee	slxe	0	0	1	34	35
41	del	slx	0	0	1	36	37
42	div	z	0	0	0	0	132
43	dmd	zs	0	0	0	0	102
44	dmu	zs	0	0	0	0	32
45	dsb	zs	0	0	0	0	31
46	dup	slxo	1	180	1	38	39
47	dus	zs	0	0	0	0	40
48	exg	zs	0	0	0	0	41
49	fad	z	0	0	0	0	181
50	fdv	z	0	0	0	0	184
51	fef	zs	0	0	0	0	113
52	fif	zs	0	0	0	0	114
53	fmu	z	0	0	0	0	183
54	fsb	z	0	0	0	0	182
55	hlt	zs	0	0	0	0	0
56	inc	z	0	0	0	0	133
57	ine	xe	0	0	1	186	187
58	inl	lx	5	188	1	193	42
59	inn	lx	0	0	1	194	43
60	ins	zs	0	0	0	0	44
61	ior	slxo	4	158	1	45	46
62	ios	zs	0	0	0	0	47
63	lab	z	0	0	0	0	0
64	lae	xe	13	100	9	113	122
65	lai	sly	0	0	1	115	116
66	lal	lx	7	92	1	99	48
67	lar	xe	0	0	2	135	137
68	las	zs	0	0	0	0	49
69	lde	xe	0	0	2	145	147
70	ldf	slx	0	0	1	50	51
71	ldl	lx	7	148	1	155	52
72	lex	slo	2	141	1	53	54
73	lin	-	0	0	1	195	196
74	lnc	lo	1	37	1	38	108
75	lni	z	0	0	0	0	185
76	loc	-	33	1	2	34	36
77	loe	xe	20	56	4	76	80
78	lof	xo	9	81	1	90	91
79	loi	ly	5	123	1	128	55
80	lol	lx	16	39	1	55	56

81	lop	lx	2	162	1	164	57
82	lor	sl	0	0	1	58	106
83	los	zs	0	0	0	0	59
84	lsa	sl	0	0	1	104	105
85	mod	z	0	0	0	0	197
86	mon	zs	0	0	0	0	60
87	mrk	slm	4	244	1	61	62
88	mrs	zsm	0	0	0	0	65
89	mrx	sla	4	248	1	63	64
90	mul	z	0	0	0	0	131
91	mxs	zsa	0	0	0	0	66
92	neg	z	0	0	0	0	198
93	nop	zs	0	0	0	0	89
94	nul	z	0	0	0	0	0
95	pad	zs	0	0	0	0	3
96	psb	zs	0	0	0	0	117
97	rck	xe	0	0	1	143	144
98	rcs	zs	0	0	0	0	118
99	res	zs	0	0	0	0	103
100	ret	slx	2	199	1	67	68
101	rol	zs	0	0	0	0	69
102	ror	zs	0	0	0	0	70
103	rtt	zs	0	0	0	0	119
104	sai	sly	0	0	1	120	121
105	sar	xe	0	0	2	201	203
106	sas	zs	0	0	0	0	71
107	sde	lxe	0	0	1	204	72
108	sdf	slx	0	0	1	73	74
109	sdl	lx	0	0	1	205	75
110	ses	zs	0	0	0	0	76
111	set	slx	0	0	1	77	78
112	shl	z	0	0	0	0	206
113	shr	zs	0	0	0	0	91
114	sig	zs	0	0	0	0	122
115	ste	xe	0	0	2	207	209
116	stf	lxo	3	210	1	213	79
117	sti	ly	2	214	1	216	80
118	stl	lx	10	217	1	227	81
119	stp	lx	0	0	1	228	109
120	str	sl	0	0	1	82	107
121	sts	zs	0	0	0	0	83
122	sub	z	0	0	0	0	130
123	teq	z	0	0	0	0	230
124	tge	z	0	0	0	0	231
125	tgt	z	0	0	0	0	232
126	tle	z	0	0	0	0	233
127	tlt	z	0	0	0	0	234
128	tne	z	0	0	0	0	235
129	trp	zs	0	0	0	0	123
130	xor	slx	0	0	1	86	87
131	xos	zs	0	0	0	0	88
132	zeq	lbo	0	0	1	236	94
133	zge	lbo	0	0	1	237	95
134	zgt	lbo	0	0	1	238	96
135	zle	lbo	0	0	1	239	97

136	zlt	lbo	0	0	1	240	98
137	zne	lbo	0	0	1	241	99
138	zre	lxe	0	0	1	242	92
139	zrl	lx	0	0	1	243	93

The above three tables, together with an optimizer table, are maintained on a single file, which can be read in by all the involved programs. These tables are separated then by an empty line. See the initialize routine in appendix 1.

The table above results in the following dispatch tables. Dispatch tables are used by interpreters to jump to the routines implementing the EM-1 instructions, indexed by the next opcode. Each line of the dispatch tables gives the routine names of eight consecutive opcodes, preceded by the opcode number of the first opcode of that line. Routine names consist of an EM-1 mnemonic followed by a suffix. The suffices indicate the encoding used. The following suffices exist:

.z	instruction without operands
.l	instruction with 16-bit operand
.s<num>	shortie with <num> as high order byte
.<num>	mini with <num> as operand

First the dispatch table for the 256 primary opcodes:

0	escape;	loc.0;	loc.1;	loc.2;	loc.3;	loc.4;	loc.5;	loc.6
8	loc.7;	loc.8;	loc.9;	loc.10;	loc.11;	loc.12;	loc.13;	loc.14
16	loc.15;	loc.16;	loc.17;	loc.18;	loc.19;	loc.20;	loc.21;	loc.22
24	loc.23;	loc.24;	loc.25;	loc.26;	loc.27;	loc.28;	loc.29;	loc.30
32	loc.31;	loc.32;	loc.s0;	loc.s1;	loc.l;	lnc.1;	lnc.s0;	lol.0
40	lol.1;	lol.2;	lol.3;	lol.4;	lol.5;	lol.6;	lol.7;	lol.8
48	lol.9;	lol.10;	lol.11;	lol.12;	lol.13;	lol.14;	lol.15;	lol.s0
56	loe.0;	loe.1;	loe.2;	loe.3;	loe.4;	loe.5;	loe.6;	loe.7
64	loe.8;	loe.9;	loe.10;	loe.11;	loe.12;	loe.13;	loe.14;	loe.15
72	loe.16;	loe.17;	loe.18;	loe.19;	loe.s0;	loe.s1;	loe.s2;	loe.s3
80	loe.l;	lof.1;	lof.2;	lof.3;	lof.4;	lof.5;	lof.6;	lof.7
88	lof.8;	lof.9;	lof.s0;	lof.l;	lal.0;	lal.1;	lal.2;	lal.3
96	lal.4;	lal.5;	lal.6;	lal.s0;	lae.0;	lae.1;	lae.2;	lae.3
104	lae.4;	lae.5;	lae.6;	lae.7;	lae.8;	lae.9;	lae.10;	lae.11
112	lae.12;	lae.s0;	lae.s1;	lae.s2;	lae.s3;	lae.s4;	lae.s5;	lae.s6
120	lae.s7;	lae.s8;	lae.l;	loi.0;	loi.1;	loi.2;	loi.3;	loi.4
128	loi.s0;	add.z;	sub.z;	mul.z;	div.z;	inc.z;	dec.z;	lar.s0
136	lar.s1;	lar.l;	aar.s0;	aar.s1;	aar.l;	lex.1;	lex.2;	rck.s0
144	rck.l;	lde.s0;	lde.s1;	lde.l;	ldl.0;	ldl.1;	ldl.2;	ldl.3
152	ldl.4;	ldl.5;	ldl.6;	ldl.s0;	adi.s0;	cif.z;	ior.1;	ior.2
160	ior.3;	ior.4;	lop.0;	lop.1;	lop.s0;	beq.s0;	bge.s0;	bgt.s0
168	ble.s0;	blt.s0;	bne.s0;	brb.s0;	brf.s0;	blm.s0;	cal.s0;	cfi.z
176	cmf.z;	cmi.z;	csa.z;	csb.z;	dup.1;	fad.z;	fsb.z;	fmu.z
184	fdv.z;	lni.z;	ine.s0;	ine.l;	inl.0;	inl.1;	inl.2;	inl.3
192	inl.4;	inl.s0;	inn.s0;	lin.s0;	lin.l;	mod.z;	neg.z;	ret.0

```
200   ret.1;  sar.s0; sar.s1; sar.l;  sde.s0; sdl.s0; shl.z;  ste.s0
208   ste.s1; ste.l;  stf.1;  stf.2;  stf.3;  stf.s0; sti.0;  sti.1
216   sti.s0; stl.0;  stl.1;  stl.2;  stl.3;  stl.4;  stl.5;  stl.6
224   stl.7;  stl.8;  stl.9;  stl.s0; stp.s0; and.1;  teq.z;  tge.z
232   tgt.z;  tle.z;  tlt.z;  tne.z;  zeq.s0; zge.s0; zgt.s0; zle.s0
240   zlt.s0; zne.s0; zre.s0; zrl.s0; mrk.0;  mrk.1;  mrk.2;  mrk.3
248   mrx.0;  mrx.1;  mrx.2;  mrx.3;  beg.1;  beg.2;  beg.3;  beg.s0
```

Next, the list of secondary opcodes:

```
  0   hlt.z;  aas.z;  adi.l;  pad.z;  and.s0; and.l;  ans.z;  beg.l
  8   beq.l;  bge.l;  bgt.l;  ble.l;  blt.l;  bne.l;  brf.l;  brb.l
 16   bes.z;  blm.l;  bls.z;  cal.l;  cas.z;  cdi.z;  cid.z;  cmd.z
 24   cms.z;  cmu.s0; cmu.l;  com.s0; com.l;  cos.z;  dad.z;  dsb.z
 32   dmu.z;  ddv.z;  dee.s0; dee.l;  del.s0; del.l;  dup.s0; dup.l
 40   dus.z;  exg.z;  inl.l;  inn.l;  ins.z;  ior.s0; ior.l;  ios.z
 48   lal.l;  las.z;  ldf.s0; ldf.l;  ldl.l;  lex.s0; lex.l;  loi.l
 56   lol.l;  lop.l;  lor.s0; los.z;  mon.z;  mrk.s0; mrk.l;  mrx.s0
 64   mrx.l;  mrs.z;  mxs.z;  ret.s0; ret.l;  rol.z;  ror.z;  sas.z
 72   sde.l;  sdf.s0; sdf.l;  sdl.l;  ses.z;  set.s0; set.l;  stf.l
 80   sti.l;  stl.l;  str.s0; sts.z;  cmp.z;  illins; xor.s0; xor.l
 88   xos.z;  nop.z;  illins; shr.z;  zre.l;  zrl.l;  zeq.l;  zge.l
 96   zgt.l;  zle.l;  zlt.l;  zne.l;  cdf.z;  cfd.z;  dmd.z;  res.z
104   lsa.s0; lsa.l;  lor.l;  str.l;  lnc.l;  stp.l;  illins; illins
112   illins; fef.z;  fif.z;  lai.s0; lai.l;  psb.z;  rcs.z;  rtt.z
120   sai.s0; sai.l;  sig.z;  trp.z;  illins; illins; illins; illins
```

16. APPENDIX 3. AN EXAMPLE PROGRAM

```
1       program example(output);
2       {This program just demonstrates typical EM-1 code.}
3       type rec = record r1: integer; r2:real; r3: boolean end;
4       var mi: integer;  mx:real;  r:rec;
5
6       function sum(a,b:integer):integer;
7       begin
8         sum := a + b
9       end;
10
11      procedure test(var r: rec);
12      label 1;
13      var i,j: integer;
14          x,y: real;
15          b: boolean;
16          c: char;
17          a: array[1..100] of integer;
18
19      begin
20              j := 1;
21              i := 3 * j + 6;
22              x := 4.8;
23              y := x/0.5;
24              b := true;
25              c := 'z';
26              for i:= 1 to 100 do a[i] := i * i;
27              r.r1 := j+27;
28              r.r3 := b;
29              r.r2 := x+y;
30              i := sum(r.r1, a[j]);
31              while i > 0 do begin j := j + r.r1; i := i - 1 end;
32              with r do begin r3 := b;  r2 := x+y;  r1 := 0 end;
33              goto 1;
34      1:      writeln(j, i:6, x:9:3, b)
35      end; {test}
36      begin {main program}
37        mx := 15.96;
38        mi := 99;
39        test(r)
40      end.
```

The EM-1 code as produced by the Pascal-VU compiler is given below. Comments have been added manually. Note that this code has already been optimized.

```
   hol 542              ; Uninitialized externals and bufs occupy 542 bytes
   pro sum,4,1          ; Procedure sum; 4 bytes of parameters & externals
   fwc .1               ; .1 will be defined later
   beg 2                ; Ask room for function result
   lin 8                ; Code from source line 8
   ldl 0                ; Load two locals ( a and b )
   add                  ; Add them
   ret 2                ; Return result
   let .1,2             ; Now define .1
   end                  ; End of procedure
   mes 3,218,100        ; compiler temporary not referenced indirect
   mes 3,16,0           ; the same is true for i, j, b and c in test
   mes 3,14,0           ; the optimizer moves these pseudos to the
   mes 3,4,0            ; start of the procedure
   mes 3,2,0
  .2 rom 1,99,2         ; Descriptor of array a[]
   pro test,2,1         ; the compiler exports all level 0 procedure names
   fwc .3
   beg 218              ; Room for local variables
   lin 20               ; Maintain source line number
   loc 1
   stl 4                ; j := 1
   lni                  ; Was a lin 21 prior to optimization
   loc 3
   lol 4
   mul
   loc 6
   add
   stl 2                ; i := 3 * j + 6
   lni                  ; Was a lin 22 prior to optimization
  .4 rom 4.8            ; Assemble 4.8 in external area
   lde .4
   sdl 6                ; x := 4.8
   lni                  ; Was a lin 23 prior to optimization
   ldl 6
  .5 rom 0.5
   lde .5
   fdv
   sdl 10               ; y := x/0.5
   lni                  ; Was a lin 24 prior to optimization
   loc 1
   stl 14               ; b := true
   lni                  ; Was a lin 25 prior to optimization
   loc 122
   stl 16               ; c := 'z'
   lni                  ; Was a lin 26 prior to optimization
   loc 1
   stl 2                ; for i:= 1
  2
   lol 2
   dup 2
   mul                  ; i*i
```

```
 lal 18
 lol 2
 sar .2              ; a[i] :=
 lol 2
 loc 100
 beq 3               ; to 100 do
 inl 2
 brb 2               ; Increment i and loop
3
 lin 27
 lol 4
 loc 27
 add                 ; j+27
 stp 0               ; r.r1 :=
 lni                 ; Was a lin 28 prior to optimization
 lol 14              ; b
 lol 0
 stf 6               ; r.r3 :=
 lni                 ; Was a lin 29 prior to optimization
 ldl 6
 ldl 10
 fad                 ; x+y
 lol 0
 sdf 2               ; r.r2 :=
 lni                 ; Was a lin 30 prior to optimization
 mrk 1               ; Mark for subsequent cal
 lop 0               ; r.r1
 lal 18
 lol 4
 lar .2              ; a[j]
 cal sum             ; cal now
 stl 2               ; i :=
4
 lin 31
 lol 2
 zle 5               ; while i > 0 do
 lol 4
 lop 0
 add
 stl 4               ; j := j + r.r1
 del 2               ; i := i - 1
 brb 4               ; loop
5
 lin 32
 lol 0
 stl 218             ; Make copy of address of r
 lol 14
 lol 218
 stf 6               ; r3 := b
 ldl 6
 ldl 10
 fad
 lol 218
 sdf 2               ; r2 := x+y
 loc 0
```

```
 stp 218                ; r1 := 0
 lni                    ; Was a lin 33 prior to optimization
1                       ; Note the absence of the unnecessary goto
 lin 34
 mrk 0
 lae 14                 ; Address of output structure
 lol 4
 cal _wri               ; Write integer with default width
 mrk 0
 lae 14
 lol 2
 loc 6
 cal _wsi               ; Write integer width 6
 mrk 0
 lae 14
 ldl 6
 loc 9
 loc 3
 cal _wrf               ; Write fixed format real, width 9, precision 3
 mrk 0
 lae 14
 lol 14
 cal _wrb               ; Write boolean, default width
 mrk 0
 lae 14
 cal _wln               ; Writeln
 let .3,218
 ret 0                  ; Return without leaving anything on the stack
 end
 pro _main,0,1          ; Main program
 fwa .6
 mrk 0
 lae .6                 ; Description of external files
 lae 0                  ; Base of HOL area to relocate buffer addresses
 cal _ini               ; Initialize files, etc...
 fwc .7
 lin 37
.8 rom 15.96
 lde .8
 sde 2                  ; x := 15.96
 lni                    ; Was a lin 38 prior to optimization
 loc 99
 ste 0                  ; mi := 99
 lni                    ; Was a lin 39 prior to optimization
 mrk 0
 lae 6                  ; Address of r
 cal test
 let .7,0
.6 con 2,-1,14          ; Description of external files
 mrk 0
 loc 0                  ; Normal exit
 cal _hlt               ; Cleanup and finish
 mes 4,40               ; Length of source file is 40 lines
 mes 5                  ; Reals were used
 eof                    ; End of this input module
```

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.

```
172 000 159 250 030 002 164 245 003 115 117 109 004 001 157 242
001 007 002 073 008 071 000 003 100 002 162 242 001 002 152 163
003 218 100 255 163 003 016 000 255 163 003 014 000 255 163 003
004 000 255 163 003 002 000 255 242 002 165 001 099 002 255 164
245 004 116 101 115 116 002 001 157 242 003 007 218 073 020 076
001 118 004 075 076 003 080 004 090 076 006 003 118 002 075 242
004 165 247 003 052 046 056 255 069 242 004 109 006 075 071 006
242 005 165 247 003 048 046 053 255 069 242 005 050 109 010 075
076 001 118 014 075 076 122 118 016 075 076 001 118 002 182 080
002 046 002 090 066 018 080 002 105 242 002 080 002 076 100 008
003 058 002 017 002 183 073 027 080 004 076 027 003 119 000 075
080 014 080 000 116 006 075 071 006 071 010 049 080 000 108 002
075 087 001 081 000 066 018 080 004 067 242 002 019 245 003 115
117 109 118 002 184 073 031 080 002 135 005 080 004 081 000 003
118 004 041 002 017 004 185 073 032 080 000 118 218 080 014 080
218 116 006 071 006 071 010 049 080 218 108 002 076 000 119 218
075 181 073 034 087 000 064 014 080 004 019 245 004 095 119 114
105 087 000 064 014 080 002 076 006 019 245 004 095 119 115 105
087 000 064 014 071 006 076 009 076 003 019 245 004 095 119 114
102 087 000 064 014 080 014 019 245 004 095 119 114 098 087 000
064 014 019 245 004 095 119 108 110 162 242 003 218 100 000 152
164 245 005 095 109 097 105 110 000 001 156 242 006 087 000 064
242 006 064 000 019 245 004 095 105 110 105 157 242 007 073 037
242 008 165 247 005 049 053 046 057 054 255 069 242 008 107 002
075 076 099 115 000 075 087 000 064 006 019 245 004 116 101 115
116 162 242 007 000 242 006 151 002 250 255 255 014 255 087 000
076 000 019 245 004 095 104 108 116 163 004 040 255 163 005 255
153
```

The portable EM-1 load format for this example program, as produced by the EM-1 assembler/loader is as follows. The words of the load file are printed in octal. The first two lines form the 16 word header. The library routines are appended to the program proper.

```
007254 000021 000000 000002 000000 000000 000000 000000
002636 000015 000034 000007 000050 000000 000000 000000
141774 112010 144201 066777 012303 155402 002271 101451
100407 134732 013222 001715 113671 014222 146670 134405
160002 021271 160572 001271 024332 101664 004543 145050
024023 062042 001645 125675 141420 024433 100434 000344
027271 152047 113671 132631 000047 000511 172671 061642
024411 011610 000256 141732 024037 004757 121051 155601
022000 125401 141416 023440 066743 033456 152155 114627
033665 000155 000511 162001 134555 021303 067770 104051
067770 003450 172211 113557 002012 002256 067770 105456
067770 143614 071370 064034 141616 111045 146032 134405
061442 002317 174271 103553 000770 177617 106405 000013
000172 177407 023776 023320 000134 000563 000334 001444
```

170052 121025 122445 024010 000242 162003 126000 000403
000344 116047 154402 015653 023115 132121 166333 024421
007234 040044 071252 000437 000402 001044 024400 003730
023115 132122 166333 024437 007234 170044 071252 001042
001002 072044 060010 033042 036000 001761 126002 022003
000002 024664 003730 177307 023115 156174 023115 001234
000333 001444 170052 057015 071400 132001 166332 174003
110450 011653 000047 143400 050447 100044 162400 021754
050447 050044 162400 040044 125000 005406 153242 023770
023623 023526 101125 023664 003325 003757 023401 002725
023770 143623 023774 023720 050447 177442 021345 125252
023447 007234 121332 000050 170124 024035 053047 001400
121332 000050 166524 174021 111047 051447 000007 166074
021004 000146 000573 151047 177707 061437 156010 076072
000137 000570 024162 000137 000570 021047 162740 060042
024252 025162 000137 000570 023515 132122 000715 075450
006354 000135 000163 027442 001252 155450 010253 025226
001234 153734 026562 000137 000570 027562 003001 036000
020361 000736 001047 000716 012054 036000 012361 002542
002054 036000 006361 125005 000411 000455 012054 036000
001754 000047 027173 101055 156664 124437 017002 025735
124002 061413 114021 000004 170474 025403 001645 000047
000573 061453 000021 000003 177402 153377 010543 000137
000570 034162 002634 034332 156664 016754 036162 000137
000570 005453 021305 100460 000050 177402 132377 153332
005453 132204 166335 125402 024027 000137 000570 162001
061403 156010 000137 000563 155664 015354 155051 075450
003354 116050 155001 005253 112451 072400 002403 036000
003400 177377 021253 000047 143400 040777 004543 025736
050042 001650 050042 172335 025625 004143 003543 007256
027337 003354 026442 000141 000170 167057 021010 060460
074000 126000 026430 166173 061006 071400 126000 021002
060460 074000 000000 004044 167457 125402 025430 003357
027042 000141 000170 004301 167057 000016 002044 170053
021010 060460 074000 125400 000023 002044 170053 026422
166173 061006 071400 126000 021002 060460 074000 125400
174030 061447 026011 004543 072400 112452 172307 003624
001656 177707 061405 156010 132050 166735 024021 100044
125000 174010 071047 003505 112451 025707 156706 005453
021305 100460 000052 177402 132377 153334 005453 132204
166335 125402 024027 005355 026442 000052 177402 132377
153334 023770 061452 025010 072400 112451 024307 153242
023770 143627 023770 001135 114451 172307 024624 127264
143423 124226 025002 172333 113224 007656 172307 003224
012656 024307 004354 023770 044562 024405 143631 023770
046162 024406 143631 023770 113013 050447 010044 117000
151047 174307 006447 023626 022121 000020 023636 143722
023770 025235 101051 025334 005757 121041 174326 111447
022000 125403 000016 001044 170051 056413 071400 121000
174326 111447 010653 174307 116447 023770 143623 150047
023447 021121 162777 125042 002245 044042 075400 050447
100044 162400 002361 060042 075400 176307 050447 167444
162777 151047 023642 000124 162003 023400 023525 101124
023664 002725 021356 116047 162016 023400 023526 002725
116047 023416 023526 002523 036000 002354 122405 022411

053047 002245 064042 075400 174307 100134 001005 144236
056370 002600 117001 177710 024410 001355 001363 021051
124517 021003 155516 003363 162001 071004 161517 000413
112235 167660 001011 002344 116401 133224 000315 112364
003543 015256 000315 116401 003633 166260 071110 116117
161520 000414 115635 130007 022354 115764 005407 134235
003543 015256 073622 005665 133635 132257 006743 030042
031601 001000 177777 161664 153014 003301 025653 047562
050234 000063 167524 061453 000014 000163 005543 074000
125400 126025 000435 112235 166660 112027 116413 000267
001046 004715 116402 166660 115411 146411 000000 003044
013653 071051 000117 161403 026014 003361 026463 001400
006343 162055 031403 047562 052000 003755 071001 153117
047562 115710 000013 167124 071044 116117 031120 052000
015757 112364 116413 061667 127011 146432 115400 127411
161664 021015 100460 005543 074000 125400 071052 116117
031520 052000 004756 071001 116117 153117 047562 031710
005743 075463 100406 153063 075463 034442 033650 030042
153063 071063 000117 167524 031416 001000 177777 161664
075414 031605 126326 021034 031461 122326 102403 001744
170454 031020 047562 052000 002357 030042 153062 116062
161401 125413 000475 153062 047562 112310 074622 071000
154051 000002 001103 000224 162161 000002 001103 000004
000417 100003 000001 000143 000002 160003 027064 000070
027060 000065 032461 034456 000066 100003 000002 177777
000016 100007 047111 052520 000124 052517 050124 052125
000000 000002 100035 020072 000000 064546 062554 000040
020072 000000 062457 061564 070057 057543 072162 071145
067562 071562 000000 074170 074170 005170 000000 067440
020156 067563 071165 062543 066040 067151 020145 000000
100004 031455 033462 034066 000000 100006 071164 062565
000000 060546 071554 000145 000050 160001 027060 031460
000000 160001 027061 000060 000004 000000 000002 000006
000004 001415 000006 001423 000012 001212 000004 001575
000002 001626 000000 000224 000004 000253 000002 000427
000002 000631 000002 000544 000002 000470 000002 001773
000012 002100 000010 001660 000004 001535 000002 001727
000006 001544 000010 001562 000006 001553 000006 001603
000002 001643 000002 001736 000014 002107 000012 002071
000006 002613 000006 002626